

TANDY 6000

XENIX
Multuser Operating System

READ ME FIRST

All computer software is subject to change, correction, or improvement as the manufacturer receives customer comments and experiences. Radio Shack has established a system to keep you immediately informed of any reported problems with this software, and the solutions. We have a customer service network including representatives in many Radio Shack Computer Centers, and a large group in Fort Worth, Texas, to help with any specific errors you may find in your use of the programs. We will also furnish information on any improvements or changes that are "cut in" on later production versions.

To take advantage of these services, you must do three things:

- (1) Send in the postage-paid software registration card included in this manual immediately. (Postage must be affixed in Canada.)
- (2) If you change your address, you must send us a change of address card (enclosed), listing your old address exactly as it is currently on file with us.
- (3) As we furnish updates or "patches", and you update your software, you must keep an accurate record of the current version numbers on the logs below. (The version number will be furnished with each update.)

Keep this card in your manual at all times, and refer to the current version numbers when requesting information or help from us. Thank you.

APPLICATIONS SOFTWARE VERSION LOG

_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

OP. SYSTEM VERSION LOG

03.00.00



Read Carefully

In order for us to notify you of modifications or updates to this program you **MUST** complete this card and return it immediately. This card gets you information only and is **NOT** a warranty registration. Register one software package per card only. The registration card is postage paid—it costs you nothing to mail.

Two change of address cards have been included so that you may continue to receive information in the event that you move. Copy all address information from the Registration Card onto them prior to sending the Registration Card. They must show your "old address" exactly as you originally registered it with us.



Software Registration Card

Cat. No. **2606402**

Version **03.00.00**

Name _____

Company _____

Address _____

City _____ Phone (____) ____ - ____

State _____ Zip _____

Change of address

NEW ADDRESS

Name _____

Company _____

Address _____

City _____ Phone (____) ____ - ____

State _____ Zip _____

OLD ADDRESS

Name _____

Company _____

Address _____

City _____ Phone (____) ____ - ____

State _____ Zip _____

Change of address

NEW ADDRESS

Name _____

Company _____

Address _____

City _____

State _____ Zip _____

OLD ADDRESS

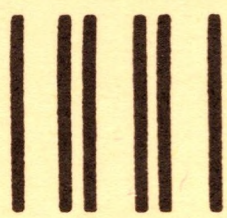
Name _____

Company _____

Address _____

City _____

State _____ Zip _____



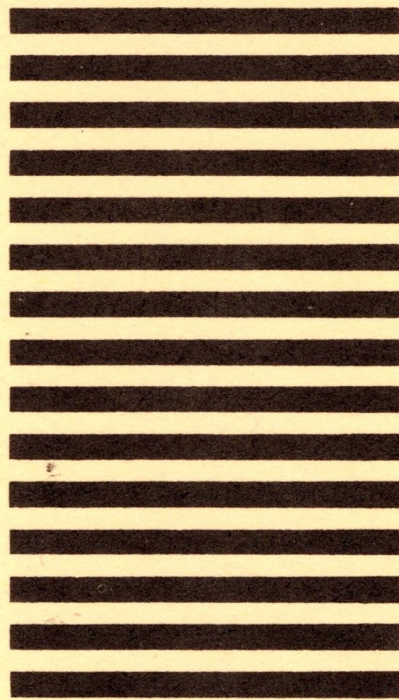
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 138 FORT WORTH, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

Software Registration
Data Processing Dept.
P.O. Box 2910
Fort Worth, Texas 76113-9965



PLACE
STAMP
HERE

Software Registration
Data Processing Dept.
P.O. Box 2910
Fort Worth, Texas 76113-2910

PLACE
STAMP
HERE

Software Registration
Data Processing Dept.
P.O. Box 2910
Fort Worth, Texas 76113-2910

Information for Software Developers

The XENIX Development System Version 3 lets you produce object modules to run on TRS-XENIX 01.03.05 (or earlier) if you install the Development 2.3 libraries. XENIX asks you if you want to install these libraries when you install the Software Development Tools of the Development system.

TRS-XENIX (versions 01.00.00 through 01.03.05) is Microsoft's Version 2.3 and is referred to as version 2. XENIX 03.00.00 and future upgrades of it are referred to as version 3.

The -v option on the C compiler (cc) command line specifies the version of object modules to be produced. If you specify -v 2, XENIX marks the object modules as version 2. If you specify -v 3 or omit -v, XENIX marks the object modules as version 3.

If you specify -v 2, note the following:

- . The symbol M_V7 is predefined by the C preprocessor. It may be used in # ifdef preprocessor directives as needed.
- . Include files are taken from /usr/include.
- . The -v 2 option passes a flag to the assembler (as), instructing it to produce version 2 linkable object files.
- . The -v 2 option also passes a flag to the loader (ld), instructing it to produce version 2 executable object files. The loader returns a nonfatal error if any version 3 object file is included in the input; whether it is specified on the command line, a module in a system library, or a user supplied library module.
- . It is particularly important that the system libraries are the correct version. The loader (ld) looks for system libraries in /usr/lib-2.3, instead of /lib and /usr/lib.
- . Most of the object modules in /usr/lib-2.3/libc.a are identical to those distributed with the TRS-XENIX Development System. However, the floating point

routines have been changed due to bugs in the 01.02.00 versions. (/usr/lib-2.3/libc.a is the system library automatically included with every program.)

- The system calls `rdchk()`, `ftime()`, `dup2()`, and those associated with semaphores and file locking are included in `/usr/lib-2.3/libc.a`. When compiling for version 2, do not use the `-lx` option on the `cc` command line if you use any of these functions. They are included in `libc.a`, and no `libx.a` library is provided.
- If you wish to install your own version 2 libraries, install them in `/usr/lib-2.3`, not `/usr/lib`. The `-lname` option on the `cc` command line refers to `/usr/lib-2.3/libname.a`. You may wish to install a corresponding version 3 library in `/usr/lib`. Do not create libraries with both version 2 and version 3 object modules in them. XENIX issues a "version mismatch" error if you try to use a mixed version library.

If you specify the `-v 3` option or omit the option, note the following:

- The symbols `M_SYS3` and `M_SYSIII` are predefined by the C preprocessor. They may be used in `# ifdef` preprocessor directives as needed.
- Include files are taken from `/usr/include`.
- The assembler produces version 3 object modules.
- The loader produces version 3 executable files.
- The loader returns a nonfatal error if any version 2 object file is included in the input; whether it is specified on the command line, a module in a system library, or a user supplied library module.
- It is particularly important that the system libraries are the correct version. The loader (`ld`) looks for system libraries in `/lib` and `/usr/lib`, not `/usr/lib-2.3`.

- . XENIX specific functions not present in Unix System III are separated into /lib/libx.a in XENIX Version 3. In order to use functions such as rdchk(), ftime(), dup2(), and functions relating to semaphores, file locking and shared data segments, use the -lx option on the cc command line. The complete list of functions kept in /lib/libx.a is:

chsize()	creatsem()	dup2()	ftime()
locking()	lock()	nap()	nbwaitsem()
opensem()	rdchk()	sdenter()	sdfree()
sdget()	sdgetv()	sdleave()	sdwaitv()
shutdn()	sigsem()	waitsem()	

- . If you wish to install your own version 3 libraries, install them in /usr/lib, not /usr/lib-2.3. The -lname option on the cc command line refers to /usr/lib/libname.a. You may wish to install a corresponding version 2 library in /usr/lib-2.3. Do not create libraries with both version 2 and version 3 object modules in them. XENIX issues a "version mismatch" error if you try to use a mixed version library.

Many of the definitions in include files have moved from one file to another, changed name slightly, or have new values. Existing programs written for version 2 may not compile due to the rearrangement. The -v 2 option does not change where the C preprocessor looks for include files. If it is necessary to compile a program with the same include files as it had on the TRS-XENIX Development System, use the "-X" and "-I/usr/include-2.3" options on the cc command line. These options are not needed if you are just linking files that have already been compiled.

Using the version 2 include files usually makes programs developed on the TRS-XENIX Development System compile properly. However, any program that makes use of data structures, particularly those defined in include files in the sys subdirectory, that have changed may not run properly under XENIX 03.00.00. This is particularly true if the

program examines data in the kernel or in disk partitions containing file systems.

Runtime Differences

Every object module, object module within an object library, and executable program is marked as either "version 2" or "version 3". During execution of a program, the behavior of certain system calls changes depending on which version of executable program is running.

The following is a list of the runtime differences between programs linked for version 2 and programs linked for version 3, running on a version 3 system:

- . In system calls where a file name is used, a zero-length file name means the current working directory in version 2, and results in an error in version 3.
- . The method of calling the following system functions has changed. If you ensure that the program is linked with the proper system runtime library, and do not try to run version 3 executable programs on a version 2 system, these changes should not affect you. The runtime library routines by the same name that exist in both versions can still be called with the same arguments as in version 2.

Changed calling sequence:

ftime()	shutdn()	locking()
creatsem()	opensem()	sigsem()
waitsem()	nbwaitsem()	rdchk()

New version 3 only calls:

setpgrp()	uname()	ustat()
fcntl()	ulimit()	chsize()
nap()	sdget()	sdfree()
sdenter()	sdleave()	sdgetv()
sdwaitv()	sigsem()	

- There are two sets of ioctl function codes for the tty driver that provide control over terminal I/O handling. The version 2 and version 3 calls are very different. In general, version 3 calls provide more explicit control over the line. Both version 2 and version 3 calls are available to programs, regardless of how the object module is marked.

Version 2 ioctl calls:

TIOCGTP
TIOCSTP
TIOCSTN
TIOCEXCL
TIOCHPCL
TIOCGETC
TIOCSETC

Version 3 ioctl calls:

TCGETA
TCSETA
TCSETAW
TCSETAF
TCFLSH

We recommend that you not mix version 2 and version 3 tty ioctl calls in the same program. The system uses version 3 calls to actually operate the serial line, and translates version 2 calls to version 3 calls. In version 2 "raw" mode, the EOF character (usually <CNTRL>-D) still exists, even though it isn't used for anything. The version 3 equivalent of "raw" mode does not have a place to store the EOF character: with "icanon" turned off, that field is used for something else. The version 2 translation code stores the EOF code in a hidden variable, and this value can pop up unexpectedly if you use a version 3 call to change modes and use a new EOF character, and then use a version 2 call to change modes. The EOF character might or might not revert back to the old one. A similar situation exists with the EOL character.

Sticking with one set of ioctl calls eliminates this problem, and makes your program more portable to other systems.

Porting C Programs

This section gives some hints on how to port C programs written for an environment similar to XENIX to your XENIX system. Some of the suggestions apply to a XENIX version 2 to a XENIX version 3 conversion, and some do not.

- . Version 3 runtime libraries have the string functions "strchr" and "strrchr", but the corresponding functions in the version 2 libraries are "index" and "rindex", respectively. If you have problems compiling or linking a program because of this, do one of the following:
 - . Compile the program in version 3 mode, and add the compiler options -Dindex=strchr -Drindex=strrchr to all compilations.
 - . Compile the program in version 2 mode, and add the compiler options -Dstrchr=index -Dstrrchr=rindex to all compilations.
 - . Use a text editor to edit your program.
- . Some compilers support the "void" type. A function returning void returns nothing useful as a return value. This type of declaration is useful to lint, so it can detect inconsistencies, such as using the return value of a function that doesn't return a value. The XENIX C compilers do not support this construct. If the program you are trying to port has void in it, do one of the following:
 - . Add an include in your program for **sys/types.h**, and use the version 3 include libraries only. This header file has the declaration "typedef int void;" in it.
 - . If none of the pieces of your program include **sys/types.h**, or you are using version 2 include libraries, add the compiler option -Dvoid=int to all compilations.
 - . Add the declaration "typedef int void;" near the beginning of any modules that get errors because of the void construct.

- . Remove all references to void with a text editor.
 - . Some compilers support very long names for variables and function names. The XENIX C compilers require that the names of external objects, including global variables and functions, be unique in the first 7 characters (8 characters in assembly language, because C adds a "_" onto the beginning of variable names).
- This does not prohibit long names, but "variable_name_1" and "variable_name_2" are not different because the first 7 characters are the same. If you try to use these in a program, XENIX might return errors from the compiler about duplicate declarations or multiple defined symbol errors from the assembler or loader. To port a program using long names, change the long names to be unique in the first 7 characters. Text editor mass substitution commands are frequently useful for this.
- . If your program is getting errors on declarations in the header files such as "ushort xyz;" inside a structure definition, you need to include **sys/types.h** before including that header file.

Using Fortran, Pascal, and Mac16 Programs

If you plan to use your TRS-XENIX version of Fortran, Pascal or Mac16 programs with the XENIX (Version 3) Development System, be sure to install the Version 2 libraries.

If you plan to execute Fortran, Pascal or Mac16 object code compiled under TRS-XENIX, log in as root and type the following commands:

```
cd /usr/lib <ENTER>
mv libp.a /usr/lib-2.3 <ENTER>
mv libf.a /usr/lib-2.3 <ENTER>
cd /usr/lib-2.3 <ENTER>
chown bin libf.a libp.a <ENTER>
```



```
chgrp bin libf.a libp.a <ENTER>
```

Next, generate an object file; then link the object file to the appropriate libraries using cc. See the examples below for more information.

Fortran Example:

If you have a Fortran source file called "sample.frt", generate an object file by typing:

```
rmfort sample -d -l <ENTER>
```

Link the object file by typing:

```
cc -v 2 -o sample sample.o -lf <ENTER>
```

Pascal Example:

If you have a Pascal source file called "sample.pas", generate an object file by typing:

```
pc -object sample.pas <ENTER>
```

Link the object file by typing:

```
cc -v 2 -o sample sample.o -lp <ENTER>
```

Mac16 Example:

If you have a Mac16 source file called "sample.src", generate an object file by typing:

```
mac16 sample <ENTER>
```

Link the object file by typing:

```
cc -v 2 -o sample sample.o <ENTER>
```


TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED
FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND RADIO SHACK FRANCHISEES OR
DEALERS AT THEIR AUTHORIZED LOCATIONS

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment. RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

- A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE." NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.**
- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY Software on **one** computer, subject to the following provisions:

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on a multiuser or network system only if either, the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

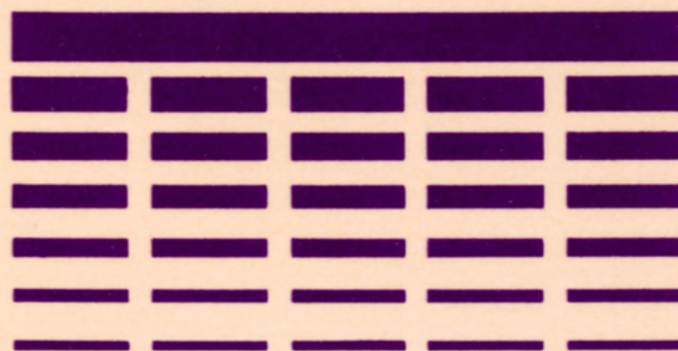
- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

XENIX™

installation guide



installation guide

TANDY®

The XENIX[®] Development System

Installation Guide

All portions of this software are copyrighted and are the proprietary and trade secret information of Tandy Corporation and/or its licensor. Use, reproduction, or publication of any portion of this material without prior written authorization by Tandy Corporation is strictly prohibited.

XENIX® Copyright 1983 Microsoft Corporation. All rights reserved. Licensed to Tandy Corporation.

XENIX® Development System Software:

Copyright 1983 Microsoft Corporation.

Copyright 1983, 84, 85 Santa Cruz Operations.

Licensed to Tandy Corporation. Portion Copyright 1985 Tandy Corporation. All rights reserved.

XENIX® Development System Installation Guide: Copyright 1985 Tandy Corporation. All rights reserved.

Reproduction or use without express written permission from Tandy Corporation of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual, or from the use of the information contained herein.

XENIX is a registered trademark of Microsoft Corporation.

UNIX is a registered trademark of Bell Laboratories.

10 9 8 7 6 5 4 3 2 1

Introduction

Introduction 1

General Tools 1

Software Development Tools 1

Text Processing Tools 1

Communications Software 1

On-Line Help Pages 1

Installing the XENIX Development System 2

Introduction

Thank you for purchasing the XENIX Development System. The Development System enhances your XENIX Runtime System to include programming tools, text-processing tools, and communication software.

The Development System is divided into packages. You may install all or some of the packages. The following is a description of each package.

General Tools

General Tools include many utilities for use with the rest of the Development System. Included are a screen editor, an alternate shell, and other tools. You must install the General Tools before any other packages in the Development System.

The General Tools occupy at least 2600 blocks of disk space (at least 1300K bytes).

Software Development Tools

Software Development Tools include utilities to aid you in many programming areas. Included are the C language compiler, a linkage editor, assembler, debugger, and libraries for source "includes" and object link modules. You must install General Tools first, before you install the Software Development Tools.

When you install the Software Development Tools, you are asked if you want to install copies of the 01.02.00 Development System include files and libraries. These are useful if you have that version of the development system and have C source code you wish to compile with minimum changes.

The Software Development Tools occupy at least 3600 blocks of disk space (at least 1800K bytes). The compatibility libraries occupy at least 500 blocks of disk space (at least 250K bytes).

Text Processing Tools

Text-Processing Tools let you enter and edit text as well as format the text for output. You can do global searches, reorganize the text ("cut and paste"), and compare text. The Text-Processing package also includes tools that let you produce output that is readable by some phototypesetters. You must install General Tools before you install the Text-Processing Tools.

The Text-Processing Tools occupy at least 3200 blocks of disk space (at least 1600K bytes).

Communications Software

The Communications Software includes the XENIX networking software and other communication tools as well as an electronic mail package. You must install General Tools before you install the Communications Software.

The Communications Software occupies at least 800 blocks of disk space (at least 400K bytes).

On-Line Help Pages

The On-line Help Pages are disk copies of most of the manual pages in the *XENIX Reference Manual* and a program to conveniently view these pages. Keyword lookup is provided.

The On-Line Help Pages occupy at least 2600 blocks of disk space (at least 1300K bytes).

Installing the XENIX Development System

Before beginning your installation, be sure that no one else is logged on the system. You must log in as root (multiuser mode) at the console to install. You must have already installed your XENIX Runtime System (Version 3.0 or later) before installing the Development System.

Now, follow these steps to install the XENIX Development System.

1. At the root prompt (#), type:

install

and press RETURN. The screen displays the Installation Menu:

Installation Menu

1. to install

q. to quit

Type *1* and press RETURN to install the XENIX Development System.

2. The screen shows:

Insert diskette in Drive 0 and press <ENTER>

Insert the INSTALL 1 disk into Drive 0 and press RETURN.

3. XENIX reads some files off the INSTALL 1 disk. The screen shows:

Remove the INSTALL 1 disk from drive 0
and press <ENTER> to continue:

Remove the disk and press RETURN.

4. The screen shows the following menu:

XENIX Development System

26-6402

INSTALLATION MENU

1. General Tools

(Installation Required)

2. Software Development Tools

3. Text Processing Tools

4. Communications Software

5. On-Line Help Pages

6. All the above

q. quit

5. Select the number corresponding to the package you want to install. Be sure to read the descriptions given earlier in this guide. Remember to install the General Tools first if you are installing individual packages of the Development System.
6. After you enter your selection, the screen shows the selection being installed and the version number. The program then displays a prompt telling you which disk(s) to insert

into Drive 0 (INSTALL 2, INSTALL 3, etc.). Follow the instructions on the screen for inserting each of the disks.

Note

Be sure to insert the correct disks when prompted. The program may ask for the disks out of numerical order.

7. XENIX begins "installing files ..." and displays the name of each file as it is installed.

8. The screen shows a message when the installation of the selected package is complete.

The screen now shows the XENIX Development System Installation Menu. You may make another selection or quit (Q). If you press Q, the system prompts you to insert the INSTALL 1 disk. Follow the instructions on the screen. The system then displays the Installation Menu. Type q, press RETURN and follow the instructions. XENIX returns you to the root prompt.

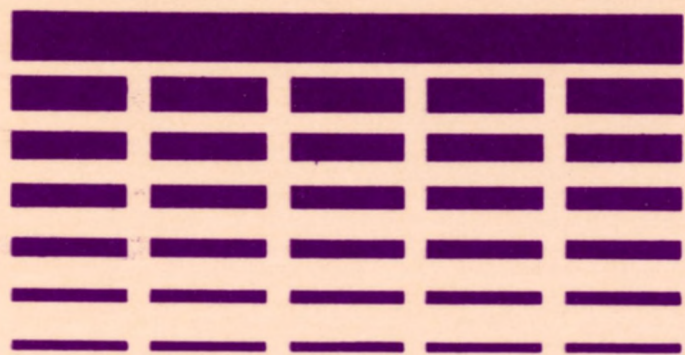
Note

1. If you have problems during installation, you may rerun the install.

2. If you didn't install all the packages, you may install them at a later date.

XENIX™

development system
operations guide



development system
operations guide

TANDY®

The XENIX®
Operating System

Operations Guide

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc. nor Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© 1983, 1984 Microsoft Corporation
© 1984, 1985 The Santa Cruz Operation, Inc.
Licensed to Tandy Corporation

XENIX is a registered trademark of Microsoft Corporation.
MS is a trademark of Microsoft Corporation.
Multiscreen is a trademark of The Santa Cruz Operation, Inc.

Document Number: G-2-14-85-1.3/1.0

Contents

1 Introduction

- 1.1 Overview 1-1
- 1.2 The System Manager 1-1
- 1.3 The Super-User Account 1-1
- 1.4 The Keyboard 1-1
- 1.5 Using This Guide 1-2

2 Starting and Stopping the System

- 2.1 Introduction 2-1
- 2.2 Starting the System 2-1
- 2.3 Logging In As the Super-User 2-2
- 2.4 Stopping the System 2-2

3 Preparing XENIX for Users

- 3.1 Introduction 3-1
- 3.2 Adding a User Account 3-1
- 3.3 Changing a User's Password 3-3
- 3.4 Forcing a New Password 3-4
- 3.5 Creating a Group 3-5
- 3.6 Changing a User's Login Group 3-6
- 3.7 Changing a User ID 3-7
- 3.8 Removing a User Account 3-8

4 Using File Systems

- 4.1 Introduction 4-1
- 4.2 File Systems 4-1
- 4.3 Permissions 4-4
- 4.4 Managing File Ownership 4-7
- 4.5 System Security 4-8
- 4.6 Using XENIX Accounting Features 4-10

5 Maintaining File Systems

- 5.1 Introduction 5-1
- 5.2 Maintaining Free Space 5-1
- 5.3 File System Integrity 5-4

6 Backing Up File Systems

- 6.1 Introduction 6-1
- 6.2 Strategies for Backups 6-1
- 6.3 Using the sysadmin Program 6-2
- 6.4 Using the tar Command 6-4

7 Using Peripheral Devices

- 7.1 Introduction 7-1
- 7.2 Adding a Terminal 7-1
- 7.3 Removing a Terminal 7-2
- 7.4 Setting the Terminal Type 7-2
- 7.5 Changing Serial Line Operation 7-3
- 7.6 Setting Serial Line Baud Rate 7-4

8 Solving System Problems

- 8.1 Introduction 8-1
- 8.2 Restoring a Nonechoing Terminal 8-1
- 8.3 Freeing a Jammed Lineprinter 8-2
- 8.4 Stopping a Runaway Process 8-2
- 8.5 Replacing a Forgotten Password 8-2
- 8.6 Removing Hidden Files 8-3
- 8.7 Restoring Free Space 8-3
- 8.8 Restoring Lost System Files 8-3
- 8.9 Restoring an Inoperable System 8-3
- 8.10 Recovering from a System Crash 8-3
- 8.11 Changing XENIX Initialization 8-4

9 Building a Micnet Network

- 9.1 Introduction 9-1
- 9.2 Planning a Network 9-1
- 9.3 Building a Network 9-4
- 9.4 Starting the Network 9-8
- 9.5 Testing a Micnet Network 9-9
- 9.6 Using a Uucp System 9-11

A XENIX Special Device Files

- A.1 Introduction A-1
- A.2 File System Requirements A-1
- A.3 Special Filenames A-1
- A.4 Block Sizes A-1
- A.5 Gap and Block Numbers A-2
- A.6 Terminal and Network Requirements A-2

B XENIX Directories

B.1	Introduction	B-1
B.2	The Root Directory	B-1
B.3	The <i>/bin</i> Directory	B-1
B.4	The <i>/dev</i> Directory	B-1
B.5	The <i>/etc</i> Directory	B-2
B.6	The <i>/lib</i> Directory	B-2
B.7	The <i>/mnt</i> Directory	B-2
B.8	The <i>/tmp</i> Directory	B-2
B.9	The <i>/usr</i> Directory	B-3
B.10	Log Files	B-3

Chapter 1

Introduction

1.1 Overview 1

1.2 The System Manager 1

1.3 The Super-User Account 1

1.4 The Keyboard 1

1.5 Using This Guide 2

1.1 Overview

The XENIX operating system is a powerful system of programs, which allows you to accomplish a full spectrum of tasks, from developing high-level and assembly language programs to creating, editing, and typesetting documents. To keep this powerful system running smoothly, the XENIX system requires careful control of its operation and a regular schedule of maintenance. This guide explains how to operate and maintain the XENIX operating system on your computer, ensuring maximum performance with the least number of system problems.

This guide also explains how to expand a XENIX system with a Micnet network. A Micnet network allows serial communication between other XENIX systems in your work environment. The Micnet programs and commands include the **netutil** program, which is used to install the network, and the **mail**, **rcp**, and **remote** commands, which are used to pass messages, files, and commands over the network.

See Chapter 9 "Building A Micnet Network" for a complete explanation of the facility.

1.2 The System Manager

Every XENIX system should have at least one person in charge of system maintenance and operation. In this guide, that person is called the system manager. It is the system manager's duty to ensure the smooth operation of the system and to perform tasks that require special privileges.

Depending on the size of the system and the number of users on the system, a system manager's job can be anything from a once-a-week task to a full-time job. Even if the system is small, the system manager should faithfully perform each required maintenance task, since sloppy maintenance can affect XENIX performance.

All tasks in this guide are presented from the system manager's point of view, but many can also be accomplished by ordinary users. Since some of the tasks dramatically change the system's operation, we recommend that, whenever possible, the system manager perform these tasks. This can prevent unwanted or unnecessary changes to the system.

1.3 The Super-User Account

The super-user account is a special account for performing system maintenance tasks. It gives the system manager unusual privileges that ordinary users do not have, such as accessing all files in the system, and executing privileged commands. Many of the tasks presented in this guide require that the system manager be logged in as the super-user. To do this, the system manager must know the super-user password created during the installation of the XENIX system (see the *XENIX Installation Guide*).

Users who are authorized to act as the super-user, including the system manager, should log in as the super-user only when it is necessary to perform a system maintenance task. Even if the system manager is the only person using the system, he should create a user account for himself and use it for day-to-day work, reserving the super-user account for system maintenance tasks only.

The number of individuals who are given the super-user password should be kept to a minimum. Misuse of the super-user powers by naive users can result in a loss of data, programs, and even the XENIX system itself.

1.4 The Keyboard

Many keys and key combinations have special meanings in the XENIX system. These keys and key combinations have special names that are unique to the XENIX system, and may or may not

XENIX Operations Guide

correspond to the keytop labels on your keyboard. To help you find the special keys, the following table shows which keys on a typical console correspond to XENIX system keys. A list for your particular console is in **keyboard(M)**.

In this table, a hyphen (-) between keys means "hold down the first key while pressing the second."

XENIX Name	Keytop	Action
BREAK	Delete	Stops current action and returns to the shell. This key is also called the INTERRUPT or DELETE key.
BACKSPACE	Backspace	Deletes the first character to the left of the cursor.
CNTRL-D	Ctrl-D	Signals the end of input from the keyboard; also exits current shell or initiates the "logout" procedure if the current shell is the login shell.
CNTRL-H	Erase	Deletes the first character to the left of the cursor. Also called the ERASE key.
CNTRL-Q	Ctrl-Q	Restarts printing after it has been stopped with CNTRL-S.
CNTRL-S	Ctrl-S	Stops printing at the standard output device, for example a terminal. Does not stop the program.
CNTRL-U	Ctrl-U	Deletes all characters on the current line. Also called the KILL key.
CNTRL-\	Ctrl-\	Quits current command and creates a <i>core</i> file (Recommended for debugging only). See <i>core(F)</i> for more information.
ESCAPE	Esc	Exits the current mode; for example, exits insert mode when in the editor <i>vi</i> .
RETURN	Return	Terminates a command line and initiates an action from the shell. (The RETURN key may be denoted as ENTER on your keyboard.)

Many of these special function keys can be modified by the user. See **stty(C)** for more information.

1.5 Using This Guide

The tasks presented in this guide range from simple ones requiring very little knowledge about

XENIX, to quite complex tasks requiring extensive knowledge about XENIX and your computer.

Each chapter explains the tools and knowledge you need to complete the tasks described in that chapter. In some cases you may be required to refer to other manuals, such as the XENIX *User's Guide*.

Chapter 1 introduces this guide.

Chapter 2 explains how to start and stop the XENIX system and how to log in as the super-user, the XENIX system's special system manager account.

Chapter 3 explains how to create accounts for the users who work on your system, how to assign groups, and how to manage user IDs.

Chapter 4 explains how to create and mount file systems, how to set permissions, and how to keep the system secure.

Chapter 5 explains how to maintain free space on the root file system and other file systems.

Chapter 6 explains how to create backup copies of the root file system and other file systems.

Chapter 7 explains how to add terminals.

Chapter 8 explains how to solve system problems such as a jammed lineprinter or a forgotten password.

Chapter 9 explains how to create a multiple system mailing network with Micnet.

Appendix A presents a list of the XENIX system special files, and explains how to use these files when creating and maintaining file systems.

Appendix B presents a list of commonly used XENIX directories and log files.

Chapter 2

Starting and Stopping the System

2.1 Introduction 1

2.2 Starting the System 1

2.2.1 Loading the Operating System 1

2.2.2 Cleaning the File System 1

2.2.3 Choosing the Mode of System Operation 2

2.3 Logging In As the Super-User 2

2.4 Stopping the System 2

2.4.1 Using the shutdown Command 3

2.4.2 Using the haltsys Command 3

2.1 Introduction

This chapter explains how to start and stop the XENIX system. It also explains how to log in as the super-user.

2.2 Starting the System

Starting a XENIX system requires more than just turning on the power. You must also perform a series of steps to initialize the system for operation. Starting the system requires:

- Loading the operating system
- Cleaning the file system (if the system was improperly stopped)
- Choosing the mode of system operation

The following sections describe each of these procedures.

2.2.1 Loading the Operating System

The first step in starting the system is to load the operating system from the computer's hard disk. Follow these steps:

1. Turn on power to the computer and hard disk. The computer loads the XENIX bootstrap program and displays the message:

Xenix Boot>

2. Press the RETURN key. The bootstrap program loads the XENIX operating system.

When the system is loaded, it displays information about itself and checks to see if the "root file system" (i.e., all files and directories) is clean. If it is clean, you may choose the mode of operation. If not, the system requires you to clean the file system before choosing.

2.2.2 Cleaning the File System

You must clean the file system if the system displays the message:

Proceed with cleaning (y or n)?

This message indicates that the system was not stopped properly as described in the section "Stopping the System" given later in this chapter. The XENIX operating system requires a clean file system to perform its tasks.

To clean the file system, type *y* (for "yes") and press the RETURN key. The system cleans the file system, repairing damaged files or deleting files that cannot be repaired. It reports on its progress as each step is completed. At some point, it may ask if you wish to salvage a file. Always answer by typing *y* and pressing the RETURN key.

When cleaning is complete, the system usually asks you to choose the mode of operation, but it may also display the message:

** Normal System Shutdown **

If it displays this message, you must reload the system. You can do this by resetting the computer and repeating the steps given in the previous section. For instruction on how to reset your computer, see the hardware manual provided with the computer.

2.2.3 Choosing the Mode of System Operation

You may choose the mode of XENIX operation as soon as you see the message:

Type CONTROL-d to continue with normal startup,
(or give the root password for system maintenance):

The system has two modes: *normal operation* and *system maintenance* mode. Normal operation is for ordinary work on the system. This is the mode you should choose to allow multiple users to log in and begin work. System maintenance mode is a specialized mode reserved for work to be done by the system manager. It does not allow multiple users.

To choose normal operation, press the CNTRL-D key. The system displays a startup message and begins to execute commands found in the command file */etc/rc* described in Chapter 8, "Solving System Problems." When the commands are finished, the system displays the "login:" prompt. You may then log in as a normal user, as described in the *XENIX User's Guide*, or as the super-user, as described in the next section.

To choose system maintenance mode, type the super-user's password (sometimes called the "root password") and press the RETURN key. The system displays the message of the day and the maintenance mode prompt (#). The commands in the */etc/rc* file are not executed. (Choose system maintenance mode only if you must do system maintenance work that requires all other users to be off the system.)

2.3 Logging In As the Super-User

Many system maintenance tasks, when performed during normal operation, require that you log in as the super-user. For example, you must be logged in as the super-user to stop the system.

Before you may log in as the super-user, you need the super-user password. You also need to see the "login:" message on your terminal's screen. If you do not see this message, press the CNTRL-D key until it appears.

To log in as the super-user, follow these steps:

1. When you see the "login:" message, type the super-user's login name
root
and press the RETURN key. The system asks for the super-user's password.
2. Type the super-user's password and press the RETURN key. The system does not display the password as you type it, so type each letter carefully.

The system opens the super-user account and displays the message of the day and the super-user prompt (#).

Take reasonable care when you are logged in as the super-user. In particular, you should be very careful when deleting or modifying files or directories. Avoid using wildcard designators in filenames and frequently check your current working directory. Small errors can cause annoying and unwanted changes to the system and user files. Some errors can cause irretrievable damage to a file or the system.

You can leave the super-user account at any time by pressing CNTRL-D.

2.4 Stopping the System

Stopping the XENIX system takes more than just turning off the computer. You must prepare the system for stopping by using either the **shutdown** or the **haltsys** command. The following sections describe each command.

2.4.1 Using the shutdown Command

The **shutdown** command is the normal way to stop the system and should be used whenever the system is in normal operation mode. It warns other users that the system is about to be stopped and gives them an opportunity to finish their work.

To stop the system with the **shutdown** command, follow these steps:

1. Log in as the super-user (see the section “Logging in as Super-User” in this chapter). The system opens the super-user account and displays the message of the day and the super-user’s prompt.

2. Type

`/etc/shutdown`

and press the RETURN key. The system loads the command which in turn asks for the number of minutes you wish to elapse before the computer stops:

Minutes till shutdown? (0-15):

3. Type any number from 0 to 15 and press the RETURN key. The system displays a warning message at each terminal, asking logged in users to finish their work and log out. As soon as all users are logged out or the specified time has elapsed, the system closes all accounts, displays the message:

**** Normal System Shutdown ****

and stops.

You may now turn off the computer.

2.4.2 Using the haltsys Command

The **haltsys** command may be used to halt the system immediately. In general, it should be used only when no other users are on the system or when the system is in system maintenance mode.

To stop the system with the **haltsys** command, follow these steps:

1. Log in as the super-user (not required when in system maintenance mode). The system opens the super-user account and displays the message of the day and the super-user prompt.

2. Type

`/etc/haltsys`

and press the RETURN key. The system displays the message:

**** Normal System Shutdown ****

and stops.

You may now turn off the hard disk and computer.

Chapter 3

Preparing XENIX for Users

- 3.1 Introduction 1
- 3.2 Adding a User Account 1
- 3.3 Changing a User's Password 3
- 3.4 Forcing a New Password 4
- 3.5 Creating a Group 5
- 3.6 Changing a User's Login Group 6
- 3.7 Changing a User ID 7
- 3.8 Removing a User Account 8

3.1 Introduction

User accounts help the XENIX system manager keep track of the people using the system, and control their access to the system's resources. Ideally, each user should have a user account. Each account has a unique "login name" and "password" with which the user enters the system, and a "home directory" where the user does his work.

It is the system manager's job to create accounts for all users on the system. It is also the manager's job to maintain user accounts by changing user passwords, login groups, and user IDs when necessary.

This chapter explains how to:

- Add user accounts to the system
- Change an account's password
- Force new passwords
- Create a group
- Change an account's login group
- Change an account's user ID
- Remove user accounts from the system

The following sections describe each in detail.

3.2 Adding a User Account

You may add a user account to the system with the **mkuser** program. The program creates a new entry in the XENIX system's */etc/passwd* file. This entry contains information about the new user, such as login name and initial password, that the system uses to let the user log in and begin work. The program also creates a home directory for the user, a mailbox for use with the **mail** command, and a *.profile* or *.login* file which contains XENIX commands that are executed when the user logs in.

To create a new user account, follow these steps:

1. Log in as the super-user.
2. Type

mkuser

and press the RETURN key. The system displays the following message:

Newuser

Add a user to the system

Do you require detailed instructions? (y/n):

3. Type the letter *y* (for "yes"), if you want information about the program, otherwise type the letter *n* (for "no"). Type *q* (for "quit") only if you wish to stop the program and return to the system. If you type a "q" to any "(y/n)" prompt, the program will stop and no changes will be made.

When the program continues, it asks you to enter the new user's login name:

Enter new user's login name:

The login name is the name by which XENIX will know the user. It is usually a short version of the user's actual name, typed in lowercase letters. For example, either "johnd" (a first name and last initial) or "jdoe" (a first initial and last name) is acceptable for the user John Doe.

4. Type the new name, and press the RETURN key. The program now asks for information about the new user's group name and group number.

A group name is the name of the group of users to which the new user will belong. Users in a group have access to a common set of files and directories. The group name is optional. If not given, the XENIX system's common group "group" (with group ID 50) is used.

The program asks:

Do you want to use the default group? (y/n):

If you type "y" the user's group name will be "group" and group ID number will be 50.

If you type "n" the program responds with a list of existing groups:

Existing groups are:

Group "group" (50): demo vdemo cdemo

Do you want to use one of these groups? (y/n):

If you type either "y" or "n" you are asked which group you want to use. Type in the name of the group. You may create a new group by typing in the new name.

Next, you are prompted for a group number. The group ID, or number, may be any number from 50 to 30000 that isn't already used for another group.

5. After entering the group name and ID, you are prompted for the initial password.

Enter password:

The initial password is the password you assign to the new user. The user will use the initial password to enter the account for the first time. Once in the account, the user should create a new password for himself, one that is hard to guess. (See the section "Changing Your Password" in the *XENIX User's Guide*.)

6. Type the password carefully, and press the RETURN key.
7. Next, you are asked for a shell type. You see a list of shells (**sh**, **vsh**, **cs**) and the prompt:

ENTER Shell type (1, 2, or 3) and press ENTER:

sh is the standard shell. **vsh** is the menu driven "visual" shell and **cs** is the c-shell.

8. Type the desired shell number and press RETURN. After you have entered the shell type, the program asks for a comment:

Please enter Comment >-----
>

A comment is information about the new user, such as a department name and phone extension. Although, the comment is optional, it is useful if the **finger** command is often used to display information about users. If given, the comment must be no more than 20 characters long, including spaces. It must not contain any colons (:). The example

John Doe, 123

shows the recommended form for a comment.

9. Type the comment. Make sure it is 20 characters or less. If you do not wish to enter a comment, just press the RETURN key.

The program now shows what you have typed and the special user entry that it has created for the new user. This entry is copied to the special system file */etc/passwd*. The entry shows the login name, the password (encrypted), the user ID, the group ID, the comment, the user's home directory, and the startup program. Items in the entry are separated by colons (:). (For a full description of each item, see *passwd(M)* in the *XENIX Reference Manual*.)

The program then gives you an opportunity to change the user name, password, group, or comment:

```
Username is "johnd", user ID is 2001.
Group name is "group", group number is 50.
Comment field is: "John Doe, 123"
Shell is "/bin/csh"
```

Do you want to change anything? (y/n):

10. Type the letter *y* (for "yes") and press the RETURN key, if you wish to change something. Type *n* (for "no") and skip to the next step if you wish to complete the new account. (Type *q*, for "quit", only if you wish to leave the program and abort the new account.)

If you type *y* the program asks for the item you wish to change:

```
username
password
group
comment
shell
```

Type the name of the item you wish to change and press RETURN. After you have changed an item, you see the complete list of items and are asked if you wish to make other changes. When you are finished with any changes, the program adds the user.

11. The program displays the message:

```
Password file updated
```

followed by a description of the actions it has taken to add the new user account to the system. The program then asks if you wish to add another user to the system.

12. Type *y* if you wish to add another user. Otherwise, type *n* to stop the program and return to the super-user prompt.

A user can log into a new account as soon as it is created. See the *XENIX User's Guide* for details.

3.3 Changing a User's Password

Normally, an ordinary user can change the password of his own account with the **passwd** command (see the *XENIX User's Guide*). Sometimes, however, it may be necessary for the super-user to change the password for him, for example, if the user has forgotten his password and cannot get into the account to change it. The super-user may change the password of any user (including himself) with the **passwd** command.

To change a password, follow these steps:

1. Log in as the super-user.
2. Type
`passwd login-name`
(where *login-name* is the user's login name) and press the RETURN key. The command displays the message:
New password:
3. Type the new password and press the RETURN key. The command does not display the password as you type it, so type carefully. The command asks you to type the password again:
Retype new password:
4. Type the password again and press the RETURN key.

To see how an ordinary user can change his own password with the *passwd* command, see the *XENIX User's Guide*.

3.4 Forcing a New Password

From time to time, a user account may need a higher level of security than ordinary. Since the security of any account depends its password, it is important to keep the password as secret as possible. One way to provide greater security is to force users to change their passwords on a regular basis.

You can force users to change their passwords by using the **pwadmin** command. This command automatically dates each password and requires the user to provide a new password when the specified number of weeks have passed. The command also requires users to wait a minimum number of weeks before allowing them to restore their previous password. To use the **pwadmin** command, you must log in as the super-user.

You can enable password aging for a specified user by using the **-a** option. Type:

```
pwadmin -a login-name
```

where *login-name* is the login name of a user. The user will then be required to wait a minimum number of weeks before he can change his password, and will be forced to change his password after a maximum number of weeks have elapsed. The **-a** option uses the default minimum and maximum values found in the */etc/default/password* file.

You can choose your own minimum and maximum number of weeks by using the **-min** and **-max** options. For example, a common pair of minimum and maximum values is 2 and 8. To set the minimum and maximum dates, type:

```
pwadmin -min num -max num login-name
```

where *num* is a number in the range 0 to 63, and *login-name* is simply the login name of the user whose password you are administering. Note that the minimum and maximum cannot both be 0, and that the minimum must not be greater than the maximum.

If you are unsure of the current minimum and maximum values for a password, you can display them by typing:

```
pwadmin -d login-name
```

This command does not change the current values.

If you wish to force a user to change his password immediately, type:

```
pwadmin -f login-name
```

The user is asked on his next login to supply a new password.

When a password no longer requires extra security, you can remove the current minimum and maximum values for the password by typing:

```
pwadmin -n login-name
```

The system will no longer prompt for changes.

For more information about password aging, see *pwadmin*(C) and *passwd*(M) in the XENIX *Reference Manual*.

3.5 Creating a Group

A group is a collection of users who share a common set of files and directories. The advantage of groups is that users who have a common interest in certain files and directories can share these files and directories without revealing them to others. Initially, all users belong to the common system group named "group", but you can create new groups by modifying the XENIX system file */etc/group* using a XENIX text editor.

To create a new group, you need to choose a group name and a group identification number (group ID). You also need to make a list of the users in the new group. The group name may be any sequence of letters and numbers up to eight characters long, and the group ID may be any number in the range 50 to 30000. Both the group name and ID must be unique, i.e., they must be not be the same as any existing group name or ID.

To create a new group, follow these steps:

1. Log in as the super-user.
2. Display the contents of the */etc/group* file by typing:

```
cat /etc/group
```

and pressing the RETURN key. The **cat** command displays the contents of the */etc/group* file. The file contains several entries, each defining the group name, group ID, and users for a group. Each entry has the form:

```
group-name::group-ID:users
```

The users are shown as a list of login names separated by commas (.). For example, a typical file may look like this:

```
other:x:1:demo
sys:x:2:
group::50:johnd,suex
```

3. Check the */etc/group* file entries to see that the group name and ID you have chosen are unique.
4. If the group name and ID are unique, invoke a XENIX text editor (see the XENIX *User's Guide*) and specify */etc/group* as the file to edit.
5. Locate the last line in the file, then insert the new entry in the form given above. For example, if you wish to create a group named "shipping" with group ID "142" and users "johnd", "marym", and "suex", type:


```
shipping::142:johnd,marym,suex
```

6. Exit the editor.

To make sure you have entered the group names correctly, use the **grpcheck** command to check each entry in the */etc/group* file. If the new entry is free of errors, no other changes to the file are required.

You can create any number of new groups. Each group may have any number of members. Furthermore, any user may be a member of any number of groups. Multiple group membership is especially convenient for users who have interests that span a variety of areas.

If a user is a member of several groups, he can gain access to each group by using the **newgrp** command. See the *XENIX User's Guide* for details.

3.6 Changing a User's Login Group

When a user logs in, the system automatically places the user in his "login group". This is the group given by the group ID in the user's */etc/passwd* file entry (see the section "Adding a User Account" in this chapter). You can change the user's login group by changing the group ID. To change the group ID you need the group ID of the new login group, and you need to know how to use a XENIX text editor (see the *XENIX User's Guide*).

To change the group ID, follow these steps:

1. Log in as the super-user.
2. Use the **cd** command to change the current directory to the */etc* directory. Type:

```
cd /etc
```
3. Use the **cp** command to make a copy of the */etc/passwd* file. Type:

```
cp passwd passwd+
```
4. Invoke a text editor and specify */etc/passwd+* as the file to edit.
5. Locate the desired user's password entry. Each entry begins with the user's login name.
6. Locate the user's group ID number in the user's password entry. It is the fourth item in the entry. Items are separated by colons (:). For example, the entry

```
marym:9iKlwp:205:50:Mary March, 122:/usr/marym:/bin/sh
```

has group ID "50".
7. Delete the old group ID and insert the new one. Be sure you do not delete any other portion of the user's password entry.
8. Exit the editor.
9. Use the **mv** command to save the old */etc/passwd* file. Type:

```
mv passwd passwd-
```
10. Use the **mv** command to make the edited file the new */etc/password* file. Type:

```
mv passwd+ passwd
```

You can make sure you have entered the new login group correctly by using the **pwcheck** command. If the new entry is correct, no other changes to the file are required.

You must not change the group IDs for system accounts such as “cron” and “root”. System accounts are any accounts whose user IDs are less than 200. The group ID is the fourth item in the password entry.

Note that changing a user’s login group does not change the “group ownership” of his files. Group ownership defines which group has access to a user’s files. If users in the new group wish to access the user’s files, you must change the group ownership with the **chgrp** (for “change group”) command. For details, see the section “Changing Group Ownership” in Chapter 4.

3.7 Changing a User ID

Sometimes it is necessary to change the user ID in a user’s account entry to allow a user to access files and directories transferred from other computers. In particular, if a user has different accounts on different computers and frequently transfers files and directories from one computer to another, then the user IDs in each of his account entries must be made the same. You can make them the same by modifying the account entries in the */etc/passwd* file.

To change a user ID, follow these steps at every computer for which the user has an account:

1. Log in as the super-user.
2. Use the **cd** command to change the current directory to the */etc* directory. Type:
`cd /etc`
3. Use the **cp** command to make a copy of the */etc/passwd* file. Type:
`cp passwd passwd+`
4. Invoke a XENIX text editor and specify */etc/passwd+* as the file to edit.
5. Locate the user’s account entry. Each entry begins with the user’s login name.
6. Locate and substitute the current user ID. The ID is the third item in the entry. For example, the entry
`marym:9iKlwp:205:50:Mary March, 122:/usr/marym:/bin/sh`
has user ID “205”.
7. Exit the text editor.
8. Use the **mv** command to save the old */etc/passwd* file. Type:
`mv passwd passwd-`
9. Use the **mv** command to make the edited file the new */etc/passwd* file. Type:
`mv passwd+ passwd`

No other changes to the file are required.

In most cases, you can change the user ID to the same number as the user’s most-used account. But the new number must be unique at every system for which the user has an account. If there is any conflict (for example, if the number already belongs to another user on one of the systems), you must choose a new number. You can choose any number greater than 200. Just make sure it is unique, and that you copy it to all systems on which the user has an account.

Once a user’s ID has been changed, you must change the “user ownership” of the user’s files and directories from the old user ID to the new one. You can do this with the **chown** (for “change owner”) command described in Chapter 4, “Using File Systems.” For example, to change the

ownership of johnd's home directory, type:

```
chown johnd /usr/johnd
```

Note that you may use the **find** command described in Chapter 6, "Backing Up File Systems," to locate all files and directories with the user's old user ID.

3.8 Removing a User Account

It is sometimes necessary to remove a user account from the system. You can remove a user account with the **rmuser** program. The program deletes the user's entry from the */etc/passwd* file and removes the user's home directory and mailbox.

Before you can remove the user account, you must remove all files and directories from the user's home directory, or move them to other directories. If you wish to save the files, you may use the **tar** command to copy the files to a floppy disk (see the section "Copying Files to a tar Disk" in Chapter 6).

To remove a user account, follow these steps:

1. Log in as the super-user.
2. Type:

```
cd /usr/login-name
```

and press the RETURN key to change to the user's home directory. The *login-name* must be the user's login name.
3. Make sure that you have made copies of all important files and directories in the user's home directory.
4. Use the **rm** (for "remove") command to remove all files and directories from the user's home directory. This includes any files that begin with a period (.). Directories can be removed by using the **-r** (for "recursive") option of the **rm** command. For example, the command

```
rm -r bin
```

removes the directory named *bin* and all files within this directory.
5. After removing all files and directories, make sure the user's mailbox is empty. Type:

```
cat /usr/spool/mail/login-name
```

and press the RETURN key, where *login-name* is the user's login name. If the mailbox contains text, then type:

```
cat /dev/null > /usr/spool/mail/login-name
```

and press the RETURN key.
6. When the user's home directory and mailbox are empty, type:

```
cd /usr
```

and press the RETURN key. The user's home directory cannot be removed until you have moved to another directory.
7. Type:

```
rmuser
```


and press the RETURN key. The program displays a message explaining how to remove a user:

```
****rmuser-remove a user from the system****
```

Press ENTER when you are ready.

The program asks for the login name of the user you wish to remove:

Enter name of id to be removed.

8. Type the user's login name. You should now see the message:

Removing user *name* from the system. CONFIRM? (y/n/q):

9. Type *y* (for "yes") to remove the user from the system. Otherwise type *n* (for "no") to stop the removal, or *q* (for "quit") to stop the program. The program removes the user's entry from the */etc/passwd* file, the user's mailbox, *.profile* file, and home directory. The program displays the message:

User *name* removed from the system

The program now gives you a chance to remove another user:

Do you want to remove another user? (y/n/q):

10. Type *y* to remove another user. Otherwise, type *n* or *q* to stop the program.

Note that the **rmuser** program will refuse to remove an account that has a system name, such as "root", "sys", "sysinfo", "cron", or "uucp", or a system ID (user ID below 200). Also, the program cannot remove a user account if the user's mailbox still has mail in it, or if the user's home directory contains files other than *.profile*.

Chapter 4

Using File Systems

4.1 Introduction	1
4.2 File Systems	1
4.2.1 Creating a File System	1
4.2.2 Mounting a File System	2
4.2.3 Unmounting a File System	3
4.2.4 Formatting Floppy Disks	3
4.3 Permissions	4
4.3.1 Displaying Permissions	5
4.3.2 Changing Permissions	6
4.3.3 Changing the File Creation Mask	6
4.4 Managing File Ownership	7
4.4.1 Changing User Ownership	7
4.4.2 Changing Group Ownership	8
4.5 System Security	8
4.5.1 Physical Security	8
4.5.2 Access Security	8
4.5.3 Encrypting Text Files	8
4.5.4 Protecting Special Files	9
4.5.5 Copying Floppy Disks	9
4.6 Using XENIX Accounting Features	10
4.6.1 Starting Process Accounting	11
4.6.2 Displaying Accounting Information	11

4.1 Introduction

This chapter describes one of the most important responsibilities of a system manager: controlling and recording users' access to the files and directories on the system. It introduces file systems, permissions, system security, and process accounting.

4.2 File Systems

A file system is the XENIX system's way of organizing files on mass storage devices such as hard and floppy disks. A file system consists of files, directories, and the information needed to locate and access these items.

Each XENIX system has at least one file system. This file system is called the root file system and is represented by the symbol `/`. The root file system contains all the XENIX programs that may be used by the system manager. It usually contains all the user directories as well.

A XENIX system may also have other file systems that contain user directories and application programs. One reason for using other file systems is to expand the available storage space of the system. Each additional file system adds its free space to the system's total space. New file systems can be specifically created by a user, then mounted onto the system so they can be used.

You can create additional file systems with the `mkfs(C)` command. This command creates a file system of a specified size and may also copy some files to the new system (see `mkfs(C)`). You can mount file systems with the `mount(C)` command. Once mounted, you may access the files and directories in the file systems as easily as files and directories in the root file system. (The root file system is permanently mounted.) When you are finished with a file system, you can unmount it with the `umount(C)` command.

You can create new file systems on hard and on floppy disks. A reason for creating new file systems on floppy disks is to establish a collection of application programs and data files that can be easily mounted and used when needed.

The following sections explain how to create and use file systems.

4.2.1 Creating a File System

You can create a file system on a formatted floppy disk by using the `mkfs` command. To create the file system, you need:

- A formatted floppy disk
- The special filename of a floppy disk drive
- The disk block size of the disk
- The gap and block numbers for the disk

To format a floppy disk, see the section "Formatting Floppy Disks" in this chapter. The special filenames for the disk drives, the disk block size, and the gap and block numbers depend on the specific system and are given in Appendix A.

Note that if a file system already exists on the disk, it will be destroyed by this procedure. For this reason, be particularly careful not to create a new file system on the root file system. If you destroy the root file system, you will have to reinstall the XENIX system.

To make a file system on a floppy disk, follow these steps:

1. Log in as super-user.
2. Insert a formatted floppy disk into a floppy disk drive. Make sure that the disk is not write-protected.
3. Type

/etc/mkfs specialfile blocksize gap block

(where *specialfile*, *blocksize*, *gap*, and *block* are supplied by you) and press the RETURN key.

The system automatically creates the file system. If it discovers data already on the disk, the system displays the message:

mkfs: *specialfile* contains data. Overwrite? (y/n):

If you are sure the disk contains nothing that you want to save, type *y* and press the RETURN key to overwrite the data and continue creating the file system. Otherwise, type *n*. If you type *n*, no file system is created.

For example, the following command creates a file system on the floppy disk drive */dev/fd1* with blocksize 320 and gap and block numbers 2 and 8.

/etc/mkfs /dev/fd1 320 2 8

The actual filename, blocksize, gap, and block numbers vary. See Appendix A for the information specific to your machine.

4.2.2 Mounting a File System

Once you have created a file system, you can mount it with the **mount** command. To mount a file system you need:

- The special filename of a disk drive
- The name of an empty directory

The special filenames of disk drives are given in Appendix A. The directory to receive the file system may be any directory as long as it is empty (contains no files) and is not your current working directory. Note that the directory */mnt* is specifically reserved for mounted file systems.

To mount a file system, follow these steps:

1. Log in as super-user.
2. Insert the disk containing the file system into a floppy disk drive.
3. Type the appropriate **mount** command, and press the RETURN key. The command should have the form

/etc/mount specialfile directoryname

where *specialfile* is the special filename of the disk drive containing the disk and *directoryname* is the name of the directory to receive the file system. If the disk is write-protected, make sure you include the switch “-r” at the end of the command.

For example, you may use the following command to mount the disk in disk drive */dev/fd1* onto the directory named */account*.

/etc/mount /dev/fd1 /account

Remember to make sure that the specified directory is empty before issuing the command. If the

command displays the message

mount: Structure needs cleaning

use the **fsck(C)** command to clean the file system and try to mount it again (see the section, “File System Integrity” in Chapter 5). The **mount** command displays the message

mount: Device busy

under any of these conditions: The file system has already been mounted, a user’s current working directory is in the directory onto which you wish to mount, a process in the mount directory is being executed, or any file in the mount directory is open. You cannot mount a file system if any user is in the mount directory.

To check that the file system was properly mounted, use the **cd(C)** command to change to the directory containing the mounted system and the **l(C)** command to list the contents. The command displays the files and directories in the file system. Be sure to use the **cd(C)** command to leave the directory after finishing your work in it.

Note that frequently used file systems can be mounted automatically when starting the system by appending the appropriate **mount** commands to the */etc/rc.user* file. See the section, “Changing the */etc/rc* File,” in Chapter 8 for details.

4.2.3 Unmounting a File System

You can unmount a mounted file system with the **umount** command. Unmounting a file system does not destroy its contents. It merely removes access to the files and directories in the file system.

To unmount a mounted file system, type

/etc/umount specialfile

and press the RETURN key. The *specialfile* is the name of the special file corresponding to the disk drive containing the disk with the file system. The command removes the file system from the directory on which it was mounted and makes the directory and the corresponding disk drive available for mounting another file system.

For example, the following command unmounts a file system from the disk drive */dev/fd1*:

/etc/umount /dev/fd1

Before unmounting a file system, make sure that no files or directories are being accessed by other commands or programs. The **umount** command displays the message:

umount: Device busy

if you or another user is currently in the directory containing the file system.

4.2.4 Formatting Floppy Disks

You can format floppy disks with the **diskutil(C)** program. Formatted disks are required whenever you create a file system. They are also required when you back up a file system with the **sysadmin** program (see Chapter 6, “File System Backups”).

To format a floppy disk, you must first shutdown the system. To do so by typing:

`sync; /etc/haltsys`

and press RETURN. Next, press the reset switch and the XENIX boot prompt appears on your screen. Follow these steps to format a floppy disk:

1. Beside the XENIX prompt, type:
`diskutil`
and press the RETURN key. The program displays its startup message and this prompt:
Diskutil: Hard or floppy disk (h or f)?
2. Type `f` and press the RETURN key. The screen shows:
Copy or format (c or f)?
3. Type `f` and press the RETURN key. The screen shows:
Format floppy disk in drive number (0..3)?
Respond by typing the number of the floppy drive into which you plan to place the floppy disk to be formatted. Then press the RETURN key.
4. Now the screen shows:
About to format floppy disk in drive x.
Type <enter> to proceed or <break> to abort:
5. Insert the floppy disk and press the RETURN key. Be sure the disk is not write-protected. When the floppy disk is formatted, the screen shows:
Disk format and verification complete.
About to format floppy disk in drive x.
Type <enter> to proceed or <break> to abort:
6. If you wish to format another floppy disk, remove the formatted disk, insert a new disk, and press the RETURN key. When you have formatted all the disks you will need, press the reset switch, and the XENIX boot prompt reappears on your screen. You can start up the system normally.

In general, the system manager should format spare floppy disks in advance. Note that formatting removes all data from the disk, so if you are formatting a disk that already contains data, make sure that the data is nothing you wish to save.

4.3 Permissions

Permissions control access to all the files and directories in a XENIX system. In XENIX, an ordinary user may access those files and directories for which he has permission. All other files and directories are inaccessible.

There are three different levels of permissions: user, group, and other. User permissions apply to the owner of the file; group permissions apply to users who have the same group ID as the owner;

and other permissions apply to all other users.

4.3.1 Displaying Permissions

You can display the permission settings for all the files in a directory with the **l** (for “list directory”) command. This command lists the permissions along with the name of the file’s owner, the size (in bytes), and the date and time the file was last changed. The command display has the following format:

```
-rw-rw---- 1 johnd group 11515 Nov 17 14:21 file1
```

The permissions are shown as a sequence of ten characters at the beginning of the display. The sequence is divided into four fields. The first field (the “type” field) has a single character, the other fields (“user”, “group”, and “other”, have three characters each. The characters in the fields have the following meanings.

In the “type” field:

- d** Indicates the item is a directory
- Indicates the item is an ordinary file
- b** Indicates the item is a device special block I/O file
- c** Indicates the item is a device special character I/O file

In the “user”, “group”, and “other” fields:

- r** Indicates read permission. Read permission for a file means you may copy or display the file. Read permission for a directory means you may display the files in that directory.
- w** Indicates write permission. Write permission for a file means you may change or modify the file. Write permission for a directory means you may create files or subdirectories within that directory.
- x** Indicates execute permission (for ordinary files) or search permission (for directories). Execute permission for a file means you may invoke the file as you would a program. Execute permission for a directory means you may enter that directory with the **cd** command.
- Indicates no permission.

For example, the permissions

```
-rwxrwxrwx
```

indicate an ordinary file with full read, write, and execute access for everyone (user, group, and other).

The permissions

```
-rw-----
```

indicate an ordinary file with read and write access for the user only.

The permissions

```
drwxr-x--x
```

indicate a directory with search access for everyone, read access for the user and group, and write access for only the user.

When you create a file, the XENIX system automatically assigns the following permissions:

```
-rw-r--r--
```

This means that everyone may read the file, but only the user may write to it. When you create a directory, the system assigns the permissions:

```
drwxr-xr-x
```

This means everyone may search and read the directory, but only the user may create and remove files and directories within it.

4.3.2 Changing Permissions

You can change the permissions of a file or a directory with the **chmod** (C) (for “change mode”) command. This command requires that you tell it how to change the permissions of a specific file or directory. You do so by indicating which levels of permissions you wish to change (user “u”, group “g”, or other “o”), how you wish to change them (add “+” or remove “-”), and which permissions you wish to change (read “r”, write “w”, or execute “x”).

For example, the pattern

```
u+x
```

adds execute permission for the user. The pattern

```
go-w
```

removes write permission for group and other.

The **chmod** command has the form

```
chmod pattern file ...
```

where *file* is the name of a file or directory. If more than one name is given, they must be separated by spaces. For example, to change the permissions of the file “receivables” from “-rw-r--r--” to “-rw-----”, type

```
chmod go-r receivables
```

and press the RETURN key.

After using **chmod** use the **l** command to check the results. If you have made a mistake, use **chmod** again to correct the mistake.

4.3.3 Changing the File Creation Mask

The file creation mask is a special number, kept by the system, that defines the permissions given to every file and directory created by a user. Initially, the mask has the value “022” which means every file receives the permissions

```
-rw-r--r--
```

and every directory receives the permissions

```
drwxr-xr-x
```

You can change the mask and the initial permissions your files and directories receive, by using the **umask**(C) command.

The **umask** command has the form:

```
umask value
```

where *value* is a three-digit number. The three digits represent user, group, and other permissions, respectively. The value of a digit defines which permission is given as shown by the following table:

Digit	Permission
0	Read and write (also execute for directories)
1	Read and write
2	Read (also execute for directories)
3	Read
4	Write (also execute for directories)
5	Write
6	Execute for directories
7	No permissions

For example, the command

```
umask 177
```

sets the file creation mask so that all files and directories initially have read and write permission for the user and no permissions for all others.

4.4 Managing File Ownership

Whenever a file is created by a user, the system automatically assigns “user ownership” of that file to that user. This allows the creator to access the file according to the “user” permissions. The system also assigns a “group ownership” to the file. The group ownership defines which group may access the file according to the “group” permissions. The group is the same group to which the user belongs when he creates the file.

Only one user and one group may have ownership of a file at any one time. (These are the owner and group displayed by the **l** command.) However, you may change the ownership of a file by using the **chown**(C) and **chgrp**(C) commands.

4.4.1 Changing User Ownership

You can change the user ownership of a file with the **chown** command. The command has the form:

```
chown login-name file ...
```

where *login-name* is the name of the new user and *file* is the name of the file or directory to be changed. For example, the command

```
chown johnd projects.june
```

changes the current owner of the file *projects.june* to “johnd”.

The **chown** command is especially useful after changing the user ID of a user account (see the section, “Changing a User’s ID,” in Chapter 3 of the *XENIX Operations Guide*).

The owner of a file may use **chown** to transfer ownership to another user or the super-user may use **chown** with any file.

4.4.2 Changing Group Ownership

You can change the group ownership of a file with the **chgrp(C)** command. The command has the form:

```
chgrp group-name file ...
```

where *group-name* is the name of a group given in the */etc/group* file and *files* are the name of the file you wish to change. For example, the command

```
chgrp shipping projects.june
```

changes the group ownership of the file *projects.june* to the group named "shipping".

The **chgrp** command is especially useful if you have changed the login group of a user (see the section, "Changing a User's Login Group," in Chapter 3).

4.5 System Security

Every system, no matter what its size, should have some form of protection from unauthorized access to the computer, disks, and system files. The following sections suggest ways for a system manager to protect the system.

4.5.1 Physical Security

You can protect the physical components of the computer, especially system disks, by taking these steps:

1. Keep unessential personnel out of the work area.
2. Organize and lock up all floppy disks when not in use. They should not be stored with the computer itself.
3. Keep disks away from magnetism, direct sunlight, and severe changes in temperature.
4. Do not use ball point pens to write labels on disks.
5. Make backup copies of all floppy disks (see the section, "Copying Floppy Disks," in this chapter).

4.5.2 Access Security

You can protect the system from access by unauthorized individuals by taking these steps:

1. Remind users to log out of their accounts before leaving the terminal.
2. Discourage users from choosing passwords that are easy to guess, and remind them to change passwords often.. Passwords should be at least six characters long and include letters, digits, and punctuation marks.
3. Keep the super-user password secret from all but necessary personnel.

4.5.3 Encrypting Text Files

You can usually ensure both the privacy and safety of files by setting the appropriate permissions

and maintaining system security. However, these methods cannot protect text files from unauthorized individuals who have logged in as the super-user. You can protect files from an unauthorized super-user by using the **crypt(C)** command to encrypt the file. Encryption changes the contents of the file into meaningless characters. The encryption is carried out by means of a key which you supply. The process can be reversed, and the file returned to meaningful text, by giving the same key.

For example, to encrypt the contents of the file *projects.june* and store the encrypted file in the file *projects.secret*, type:

```
crypt <projects.june >projects.secret
```

and press the RETURN key. The command asks for the key with the message:

Enter key:

Type a string of characters (it may be up to eight characters long) and press the RETURN key. The program encrypts the file.

To restore the encrypted file *projects.secret* and display it on the screen, type:

```
crypt <projects.secret
```

and press the RETURN key. The command asks for the key. Type the same key you used to encrypt the data and press the RETURN key. The program displays the restored data.

4.5.4 Protecting Special Files

You can prevent ordinary users from gaining direct access to the data and program files on the system's hard and floppy disks by protecting the system's special files. The XENIX special files, in the */dev* directory, are used primarily by the system to transfer data to and from the computer's hard and floppy disks as well as other devices, but can also be used by ordinary users to gain direct access to these devices.

Since direct access bypasses the system's normal protection mechanisms and allows ordinary users to examine and change all files in the system, it is wise to protect the special files to ensure system security.

To protect the XENIX special files, log in as the super-user and use the **chmod** command to set appropriate permissions. For example, to disallow any access by ordinary users, set the permissions of such special files as */dev/mem*, */dev/kmem*, and */dev/root*, to read and write access for the user only. Note that you must not change the permissions for the */dev/tty* files.

4.5.5 Copying Floppy Disks

To ensure against the loss of data stored on floppy disks, you can use either the **diskutil(C)** or **dd(C)** commands to make copies of floppy disks on new, formatted disks.

Note

When using **dd**, make sure there is adequate space on the *root* file system before you start. Use the **df(C)** command to check the amount of free space. With this approach, a temporary file is made in the */tmp* directory and there must be enough space to contain the file.

To make a copy of a disk with **dd**, and you have only one floppy disk drive, follow these steps:

1. Insert the disk to be copied into the floppy drive.

2. Type:

```
dd if=/dev/fd0 of=/tmp/copy count=blkcount
```

and press the RETURN key. The *blkcount* must be the number of blocks on the disk to be copied (see Appendix A for details).

Next, copy the boot track by typing:

```
dd if=/dev/fdbt0 of=/tmp/copy.boot
```

A copy of the floppy disk is made in the file */tmp/copy*. Now you can put this information back onto a different floppy.

3. Eject the floppy that is currently in the floppy drive. Insert an empty, formatted disk into the drive. If necessary, you can format a disk with the **diskutil** command described in “Formatting Floppy Disks” in this chapter.

4. Type:

```
dd if=/tmp/copy of=/dev/fd0 count=blkcount
```

```
dd if=/dev/copy.boot of=/dev/fdbt0
```

and press the RETURN key. The command copies the file */tmp/copy* to the floppy disk, then displays a record of the number of blocks copied.

You can continue to make copies by putting additional blank, formatted floppies into the drive and repeating the last step above. When you are finished you should remove the file */tmp/copy*.

If you have two floppy drives, follow these steps to copy a floppy disk:

1. Insert the disk to be copied into floppy drive 0.
2. Insert an empty, formatted disk into drive 1. If necessary, you can format a disk with the **diskutil** command described in “Formatting Floppy Disks” in this chapter.

3. Type:

```
dd if=/dev/fd0 of=/dev/fd1 count=blkcount
```

```
dd if=/dev/fdbt0 of=/dev/fdbt1
```

and press the RETURN key. The *blkcount* must be the number of blocks on the disk to be copied (see Appendix A for details).

The command copies the first disk to the second, then displays a record of the number of blocks copied.

4.6 Using XENIX Accounting Features

The XENIX system provides a set of commands that allow the system manager to perform process accounting. Process accounting is a simple way to keep track of the amount of time each user spends on the system. The process accounting commands keep a record of the number of processes (i.e., programs) invoked by a user, how long each process lasts, and other information such as how often the process accesses I/O devices, and how big the process is in bytes.

Process accounting is helpful on systems where users are being charged for their access time, but it may also be used to develop a detailed record of system, command, and system resource usage.

There are several commands which may be used to do process accounting. Of these, the most useful are **accton(C)** and **acctcom(C)**. The **accton** command starts and stops process accounting. When invoked, the command copies pertinent information about each process to the file named */usr/adm/pacct*. The **acctcom** command is used to display this information. The command has several options for displaying different types of accounting information.

4.6.1 Starting Process Accounting

Process accounting can be started at any time, but is typically started when the system itself is started. You can start process accounting with the **accton** command:

- Log in as super-user

- Type:

```
accton /usr/adm/pacct
```

The command automatically creates a new file */usr/adm/pacct*, and begins to copy process accounting information to it.

If the */usr/adm/pacct* file exists before starting **accton**, the file contents are deleted.

Note that when you start the system, the contents of the */usr/adm/pacct* file is usually saved in the file */usr/adm/opacct* by the */etc/rc*.

4.6.2 Displaying Accounting Information

The **acctcom** command reads processing accounting information from the */usr/adm/pacct* file by default, then displays selected information on your terminal screen. The command usually displays basic accounting information, such as the process's program name, the name of the user who invoked the process, the start and stop times for the process, and the number of execution seconds in real time and CPU time. The command has several options with which to select other information to display.

To display the average size of each process, type:

```
acctcom
```

The command displays the basic information plus the average size of each process.

To display basic accounting information about a specific command, type

```
acctcom -n command
```

where *command* is the name of the command you are interested in. The command responds by displaying each entry for the specified *command*. For example,

```
acctcom -n units
```

displays each entry for the system command **units**.

To display information about the number and size of input and output counts, type:

```
acctcom -i
```

The command displays basic program information plus the number of characters and blocks transferred or read by each program.

To display information about a program's use of system resources, type

```
acctcom -h
```

The command displays the basic information plus the "use factor". The use factor is a number generated and used by the system to determine how each process should be scheduled for

execution. Processes with high use factors use a high percentage of the system resources and are therefore scheduled after processes with lower factors.

Chapter 5

Maintaining File Systems

5.1 Introduction 1

5.2 Maintaining Free Space 1

5.2.1 Strategies for Maintaining Free Space 1

5.2.2 Displaying Free Space 1

5.2.3 Sending a System-Wide Message 2

5.2.4 Displaying Disk Usage 2

5.2.5 Displaying Blocks by Owner 2

5.2.6 Mailing a Message to a User 3

5.2.7 Locating Files 3

5.2.8 Locating *core* and Temporary Files 3

5.2.9 Clearing Log Files 4

5.2.10 Expanding the File System 4

5.3 File System Integrity 4

5.3.1 Repairing the File System 4

5.3.2 Automatic File System Check 5

5.1 Introduction

File system maintenance, an important task of the system manager, keeps the XENIX system running smoothly, keeps the file systems clean, and ensures adequate space for all users. To maintain the file systems, the system manager must monitor the free space in each file system, and take corrective action whenever it gets too low.

This chapter explains the file system maintenance commands. These commands report how much space is used, locate seldom-used files, and remove or repair damaged files.

5.2 Maintaining Free Space

The XENIX system operates best when at least 15% of the space in each file system is free. In any system, the amount of free space depends on the size of the disk containing the file system and the number of files on the disk. Since all disks have a fixed amount of space, it is important to carefully control the number of files stored on the disk.

If a file system has less than 15% free space, system operation usually becomes sluggish. If no free space is available, the system stops any attempts to write to the file system. This means that the user's normal work on the computer (creating new files and expanding existing ones) stops.

The only remedy for a file system which has less than 15% free space is to delete one or more files from the file system. The following sections describe strategies for keeping the free space available.

5.2.1 Strategies for Maintaining Free Space

The system manager should regularly check the amount of free space of all mounted file systems and remind users to keep their directories free of unused files. You can remind users by including a reminder in the message of the day file */etc/motd*. (See the section "Changing the */etc/motd* File" in Chapter 8).

If the amount of free space slips below 15%, the system manager should:

1. Send a system-wide message asking users to remove unused files.
2. Locate exceptionally large directories and files, and send mail to the owner asking him to remove unnecessary files.
3. Locate and remove temporary files and files named *core*.
4. Clear the contents of system log files.

Finally, if the system is chronically short of free space, it may be necessary to create and mount an additional file system.

5.2.2 Displaying Free Space

You can find out how much free space exists in a particular file system with the **df** (for "disk free") command. This command displays the number of "blocks" available on the specific file system. A block is 512 characters (or bytes) of data.

The **df** command has the form:

df *specialfile*

where *specialfile* can be the name of a XENIX special file corresponding to the disk drive containing the file system (see Appendix A, "XENIX Special Device Files"). If you do not give a

special filename, then the free space of all normally mounted file systems is given.

For example, to display the free space of the root file system */dev/root*, type:

```
df /dev/root
```

and press the RETURN key. The command displays the special filename and the number of free blocks. You may compute the percentage of free space by comparing the displayed value with the total number of blocks in the file system. See Appendix A, “XENIX Special Device Files,” for a list of the total blocks.

5.2.3 Sending a System-Wide Message

If free space is low, you may send a message to all users on the system with the **wall** (for “write to all”) command. This command copies the messages you type at your terminal to the terminals of all users currently logged in.

To send a message, type:

```
wall
```

and press the RETURN key. Type the message, pressing the RETURN key to start a new line if necessary. After you have typed the message, press the CNTRL-D key. The command displays the message on all terminals in the system. To leave the **wall** command, press the CNTRL-D key. This removes the link to other terminals.

5.2.4 Displaying Disk Usage

You can display the number of blocks used within a directory by using the **du** command. This command is useful for finding excessively large directories and files.

The **du** command has the form:

```
du directory
```

The optional *directory* must be the name of a directory in a mounted file system. If you do not give a directory name, the command displays the number of blocks in the current directory.

For example, to display the number of blocks used in the directory */usr/johnd*, type:

```
du /usr/johnd
```

and press the RETURN key. The command displays the name of each file and directory in the */usr/johnd* directory and the number of blocks used.

5.2.5 Displaying Blocks by Owner

You can display a list of users and the number of blocks they own by using the **quot** (for “quota”) command. The command has the form:

```
quot specialfile
```

The *specialfile* must be the name of the special file corresponding to the disk drive containing the file system (see Appendix A, “XENIX Special Devices Files”).

For example, to display the owners of files in the file system mounted on the disk drive */dev/fd1*, type:

```
quot /dev/fd1
```

and press the RETURN key. The command displays the users who have files in the file system and the number of blocks in these files.

5.2.6 Mailing a Message to a User

If a particular user has excessively large directories or files, you may send a personal message to the user with the **mail** command.

To begin sending a message through the mail, type

```
mail login-name
```

and press the RETURN key. The *login-name* must be the login name of the recipient. To send a message, type the message, press the RETURN key, and then press the CNTRL-D key. If the message has more than one line, press the RETURN key at the end of each line. The **mail** command copies the message to the user's mailbox, where he may view it also by using the **mail** command. See the *XENIX User's Guide* for details.

5.2.7 Locating Files

You may locate all files with a specified name, size, date, owner, and/or last access date by using the **find** command. The command is useful for locating seldom-used and excessively large files.

The **find** command has the form:

```
find directory parameters
```

The *directory* must be the name of the first directory to be searched. (It will also search all directories within that directory.) The parameters are special names and values that tell the command what to search for (see *find(C)* in the *XENIX Reference Manual* for complete details). The most useful *parameters* are:

```
-name file
```

```
-atime number
```

```
-print
```

The “-name” parameter causes the command to look for the specified *file*. The “-atime” parameter causes the command to search for files which have not been accessed for the *number* of days. The “-print” parameter causes the command to display the locations of any files it finds.

For example, to locate all files named *core* in the directory */usr*, type:

```
find /usr -name core -print
```

and press the RETURN key. The command displays the locations of all files it finds.

5.2.8 Locating *core* and Temporary Files

You can locate *core*, and temporary files with the **find** command.

A *core* file contains a copy of a terminated program. The XENIX system sometimes creates such a file when a program causes an error from which it cannot recover. A temporary file contains data created as an intermediate step during execution of a program. These files may be left behind if a program contained an error or was prematurely stopped by the user. The name of a temporary file depends on the program that created it.

In most cases, the user has no use for either *core* or temporary files and they can be safely removed.

When searching for *core* or temporary files, it is a good idea to search for files which have not been accessed for a reasonable period of time. For example, to find all *core* files in the */usr* directory which have not been accessed for a week, type:


```
find /usr -name core -atime +7 -print
```

and press the RETURN key.

5.2.9 Clearing Log Files

The XENIX system maintains a number of files, called log files, that contain information about system usage. When new information is generated, the system automatically appends this information to the end of the corresponding file, preserving the file's previous contents. This means the size of each file grows as new information is appended. Since the log files can rapidly become quite large, it is important to periodically clear the files by deleting their contents.

You can clear a log file by typing:

```
cat < /dev/null > filename
```

where *filename* is the full pathname of the log file you wish to clear. A log file normally receives information to be used by one and only one program, so its name usually refers to that program. Similarly, the format of a file depends on the program that uses it. See Appendix B, "XENIX Files and Directories," for descriptions of the log files.

In some cases, clearing a file affects the subsequent output of the corresponding program. For example, clearing the file */etc/ddate* forces the next backup to be a periodic backup (see Chapter 6, "Backing Up File Systems").

5.2.10 Expanding the File System

If free space is chronically low, it may be to your advantage to expand the system's storage capacity by creating and mounting a new file system. Once mounted, you may use this new file system for your work, or even copy user or system directories to it.

A chronic shortage of space usually results from having more users on the system than the current hard disk can reasonably handle, or having too many directories or files. In either case, creating a new file system allows some of the users and directories to be transferred from the hard disk, freeing a significant amount of space on the existing file system and improving system operation. For details about creating and mounting file systems, see Chapter 4, "Using File Systems."

5.3 File System Integrity

Since file systems are normally stored on hard and floppy disks, occasional loss of data from the file system through accidental damage to the disks is not unusual. Such damage can be caused by conditions such as an improper system shutdown, hardware errors in the disk drives, or a worn out disk.

Such damage usually affects one or two files, making them inaccessible. In very rare cases, the damage causes the entire file system to become inaccessible.

The XENIX system provides a way to restore and repair a file system if it has been damaged. The **fsck** (for "file system check") command checks the consistency of file systems and, if necessary, repairs them. The command does its best to restore the information required to access the files, but it cannot restore the contents of a file once they are lost. The only way to restore lost data is to use backup files. For details about backup disks, see Chapter 6, "Backing Up File Systems."

5.3.1 Repairing the File System

You can repair a file system with the **fsck** command. The command has the form:

`fsck specialfile`

The *specialfile* must be the name of the special file corresponding to the disk drive containing the file system (see Appendix A, “XENIX Special Device Files”).

For example, to check the file system on the disk in the disk drive `/dev/fd1`, type

```
fsck /dev/fd1
```

and press the RETURN key. The program checks the file system and reports on its progress with the following messages.

- ** Phase 1 - Check Blocks and Sizes
- ** Phase 2 - Pathnames
- ** Phase 3 - Connectivity
- ** Phase 4 - Reference Counts
- ** Phase 5 - Check Free List

If a damaged file is found during any one of these phases, the command asks if it should be repaired or salvaged. Type `y` to repair a damaged file. You should always allow the system to repair damaged files even if you have copies of the files elsewhere or intend to delete the damaged files.

Note that the **fsck** command deletes any file that it considers too damaged to be repaired. If you suspect a file system problem and wish to try to save some of the damaged file or files, check other possible remedies before you invoke the command.

5.3.2 Automatic File System Check

The XENIX system sometimes requests a check of the file system when you first start it. This usually occurs after an improper shutdown (for example, after a power loss). The file system check repairs any files disrupted during the shutdown. For details, see the section “Cleaning the File System” in Chapter 2.

Chapter 6

Backing Up File Systems

6.1 Introduction 1

6.2 Strategies for Backups 1

6.3 Using the sysadmin Program 2

6.3.1 Creating Backups 2

6.3.2 Getting a Backup Listing 2

6.3.3 Restoring a Backup File 3

6.4 Using the tar Command 4

6.4.1 Copying Files to a tar Disk 4

6.4.2 Restoring Files From a tar Disk 4

6.1 Introduction

A file system backup is a copy, on floppy disk, of the files in the root directory and other regularly mounted file systems. A backup allows the system manager to save a copy of the file system as it was at a specific time. The copy may be used later to restore files that are accidentally lost or temporarily removed from the file system to save space.

This chapter explains how to create backups of the root directory and other file systems, and how to restore files from the backups.

6.2 Strategies for Backups

The system manager should back up the root directory (and any other mounted file systems) on a regular basis. In particular, the manager should make daily copies of all files modified during the day, and should make periodic (e.g., weekly) copies of the entire root directory and other mounted file systems.

The XENIX system offers two ways to back up file systems, the **sysadmin** program and the **tar** command.

The **sysadmin** program is a formal maintenance program for systems that require a rigorous schedule of file system backups. The program automatically locates modified files, copies them to backup media, and optionally produces a list of the files. If your system has many users and a large number of files that are modified daily, use **sysadmin** to make regular backups.

The **tar** command is useful on systems with one or two users, or on any system where ordinary users wish to make personal copies of their directories and files. The command lets the system manager or user choose the files and directories to be copied. The command does not, however, automatically locate modified files.

A typical backup schedule includes a daily backup once a day and a periodic backup once a week. A daily backup copies only those files modified during that day; a periodic backup copies all files in the file system. The appropriate schedule for a system depends on how heavily the system is used and how often files are modified. In all cases, a periodic backup should be done at least once a month.

The system manager should schedule backups at times when few (if any) users are on the system. This ensures that the most recent version of each file is copied correctly.

A regular schedule of backups requires a large number of floppy disks and adequate storage for the disks. Daily backups should be saved at least two weeks; periodic backups should be saved indefinitely. Disks should be properly labeled with the date of the backup and the names of the files and directories contained in the backup. After a backup has expired, the disk may be used to create new backups.

Note

If the number of floppies needed for making backups grows too large, the system manager can use the **backup** command instead of **sysadmin**. **backup** is called by **sysadmin** and by using **backup** directly, the system administrator can tailor the number of floppies used to fit the needs of the individual site. Refer to **backup(C)** in the *XENIX Reference*.

6.3 Using the sysadmin Program

The **sysadmin** program performs daily and periodic backups, lists backup files, and restores individual files from backup disks. The program presents each task as an item in a menu. To perform a task, simply choose the appropriate item from the menu and supply the required information.

6.3.1 Creating Backups

To create backups with the **sysadmin** program, you need several formatted floppy disks. The exact number depends on the number of files to be copied; for example, some periodic backups require as many as nine disks. For details on how to format a floppy disk, see the section "Formatting Floppy Disks" in Chapter 4.

To create a backup, follow these steps:

1. Log in as the super-user.

2. Type:

sysadmin

and press the RETURN key. The program displays a file system maintenance menu.

File System Maintenance

Type	1 to do daily backup
	2 to do a periodic backup
	3 to get a backup listing
	4 to restore a file
	5 to quit

3. Type *1* for a daily backup or *2* for a periodic backup. Then press the RETURN key. Note that if the system has never had a periodic backup, it automatically performs one, even if you have chosen a daily backup.
4. Insert a floppy disk in drive 1, wait for the drive to accept the disk (all drive noise should stop), and press the RETURN key. The system displays the current date and the date of the last backup (it displays "the epoch" if there has been no backup). The system then begins to copy files to the floppy disks. If the disk runs out of space, the program displays the message:

Change volumes

5. Remove the first disk and insert a new disk. Wait for the drive to accept the disk, then press the RETURN key. The program continues to copy files to the new disk. Repeat this step until the program displays the message:

DONE

When doing a periodic backup, you may need to repeat the last step several times before the backup is complete. You should label each disk as you remove it from the disk drive. For example, label the first disk "Volume 1", the second "Volume 2", and so on.

6.3.2 Getting a Backup Listing

You can keep a record of the files you have backed up by invoking the **sysadmin** program and selecting the third item in the menu. The program copies the names of all files from the backup

disks to the temporary file */tmp/backup.list*. This listing is especially convenient if you keep detailed records of the files copied in each backup. The backup listing is available after every daily or periodic backup.

To get the listing, follow these steps:

1. Log in as the super-user.
2. Type
 sysadmin
and press the RETURN key. The program displays the system maintenance menu.
3. Type 3 and press the RETURN key. The program asks you to reinsert the backup disks in the same order that you inserted them during the backup.
4. Insert the first disk, wait until the drive accepts the disk, then press the RETURN key. The program automatically reads the filenames off the backup disk and places them in the list file. When the program has read all the names, it asks for the next disk.
5. Remove the first disk and insert the next. Wait for the drive to accept the disk and press the RETURN key. Repeat this step until all disks have been read.

You may produce a printed copy of the backup list by printing the list at the lineprinter. Type

```
lpr /tmp/backup.list
```

and press the RETURN key. To save space after printing the file, you should remove it from the */tmp* directory with the **rm** command.

6.3.3 Restoring a Backup File

You can restore files from the backup disks by invoking the **sysadmin** program and selecting the fourth item in the menu. You will need the complete set of backup disks containing the latest version of the file you wish to restore. You will also need the “full pathname” of the file you wish to restore. This is the name given for the file in the backup listing.

To restore a file, follow these steps:

1. Log in as the super-user.
2. Type
 sysadmin
and press the RETURN key. The program displays the file system maintenance menu.
3. Type 4 and press the RETURN key. The program asks you to type the full pathname of the file you wish to restore.
4. Type the pathname and press the RETURN key. The program asks for another pathname.
5. Repeat step 4 to enter another pathname, or press the RETURN key to continue the program. If you press the RETURN key, the program asks you to insert the first disk in

the backup set.

6. Insert the first disk in the set of backup disks (volume 1), wait for the drive to accept the disk, and press the RETURN key. The program displays the inode numbers of the files you have given, then asks for the volume number of the backup disk containing the files.
7. Insert the disk having the correct volume number, type the volume number, and press the RETURN key. The program searches the disk for the specified files. If found, the files are copied to your current directory. If not found, the program asks for the next volume.
8. Repeat step 7 until all files have been found and copied.

The **sysadmin** program does not restore the file's original name. Instead, it names the file a unique number called an "inode" number. You can restore the file's original name using the **mv** (for "move") command:

```
mv inode filename
```

inode is the name given to the file by **sysadmin**. *filename* is the new name you want for the file. For example, to restore a file */usr/johnd/projects.june* from 224, type:

```
mv 224 /usr/johnd/projects.june
```

6.4 Using the tar Command

The **tar** command copies specified files and directories to and from floppy disks. On systems with one or two users, it gives the system manager a direct way to make backup copies of the files modified during a day. On systems with many users, it gives ordinary users a way to make personal copies of their own files and directories.

6.4.1 Copying Files to a tar Disk

You can copy a small number of files or directories to a floppy disk with the **tar** command. The command has the form:

```
tar cvf specialfile files
```

The *specialfile* must be the name of the special file corresponding to a disk drive (see Appendix A, "XENIX Special Device Files"). The drive must contain a formatted disk. The *files* are the names of the files or directories you wish to copy.

To use the **tar** command, you need a formatted floppy disk and the names of the files and/or directories you wish to copy. For details about how to format a disk, see the section "Formatting Floppy Disks" in Chapter 4. If you give a directory name, the command copies all files in the directory (including subdirectories) to the disk.

For example, to copy the files *a*, *b*, and *c* to the disk in the disk drive */dev/fd1*, type

```
tar cvf /dev/fd1 a b c
```

and press the RETURN key.

6.4.2 Restoring Files From a tar Disk

You may also use the **tar** command to restore files from a disk. The command simply copies all files on the disk to your current directory. In this case, the command has the form:


```
tar xvf specialfile
```

The *specialfile* must be the name of the special file corresponding to the disk drive containing the **tar** disk.

For example, to restore files from the disk in the drive */dev/fd1*, type

```
tar xvf /dev/fd1
```

and press the RETURN key. The command copies files on the disk in the drive to the current directory.

Since the **tar** command copies files only to the current directory, make sure you are in the desired directory before you invoke the command. You can change to the desired directory with the **cd** command.

Chapter 7

Using Peripheral Devices

7.1 Introduction 1

7.2 Adding a Terminal 1

7.3 Removing a Terminal 2

7.4 Setting the Terminal Type 2

7.5 Changing Serial Line Operation 3

7.6 Setting Serial Line Baud Rate 4

7.1 Introduction

One important task of the system manager is to add peripheral devices, such as terminals, to the system.

To add a peripheral device, the system manager must make the physical connection between the device and the computer, then use the correct system commands to enable the device for operation. This chapter explains how to use system commands to enable a device for use.

Note that physical connections between a device and the system vary. For information about these connections, see the hardware manual provided with the device.

Make sure that any serial wire is connected to your modem, terminal, printer, etc. or not connected to your computer at all. An unterminated wire connected to your computer can considerably reduce system performance. Always disconnect unused lines at your computer, not at the device.

Note

If the device you wish to add isn't discussed in this chapter, refer to the XENIX *Installation Guide* chapter on "Adding Peripheral Devices."

7.2 Adding a Terminal

Many different terminals work well with the XENIX operating system. A short list of recommended terminals is given in **terminals(M)** in the XENIX *Reference*.

Before you can add a terminal, you must know how to connect the terminal to a serial line on the computer. Physical connections for the terminal are usually explained in the terminal's hardware manual. The names of the available serial adaptor ports are given in Appendix A of this guide. Once a terminal has been connected, you may then enable the terminal for use with the **enable(C)** command.

To add a terminal, follow these steps:

1. Using the recommended procedure in the terminal's hardware manual, connect the terminal to one of the RS-232 serial lines on the computer itself. Make sure that the terminal is compatible with the line configuration (for a description of the serial ports, see Appendix A). A null modem wire (in which pins 2 and 3 are reversed) may be necessary.
2. Log in as the super-user on the console. Plug in the terminal and turn it on.
3. Use the **enable** command to enable the terminal.

When using the **enable** command, make sure that you wait a full minute between each use of the command. Failure to do so can cause a system crash.

The command has the form:

`enable specialfile`

where *specialfile* is the name of the serial line to which the terminal is attached.

`enable /tty01`

enables the terminal connected on serial line */tty01*. Likewise, to **enable** an alternate

serial port, type:
enable /tty02

The device names used above are examples. For a listing of the devices available on your computer, see Appendix A of the XENIX *Operations Guide*, "Special Device Files."

4. Press the RETURN key on the new terminal several times. The system should display a "login:" message. When it does, you may log in and begin work.

If no "login:" message appears on the screen, if random characters appear, or if the terminal does not respond to your attempt to log in, you may need to change the baud rate (or "line speed") of the terminal to match the serial line. The default baud rate is 9600, unless altered by your system administrator. Check the rate and parity scheme recommended for the terminal, then check to see how the serial line is set by using the **stty(C)** command on the serial line in question. You can change the baud rate until you log out with the **stty(C)** command described in the section "Changing Serial Line Operation." Permanent changes are made by editing */etc/ttys* as described in the section "Setting Serial Line Baud Rate."

7.3 Removing a Terminal

From time to time it may be necessary to remove a terminal from the system, for example, if you wish to replace it with a serial line printer. Before you remove a terminal, you must disable it with the **disable(C)** command.

To remove a terminal, follow these steps:

1. Log in as the super-user on the console.
2. Use the **disable** command to disable the terminal.

When using the **disable** command, make sure that you wait a full minute between each use of the command. Failure to do so can cause a system crash.

The command has the form:

```
disable specialfile
```

where *specialfile* is the name of the serial line to which the terminal is attached. For example, the command

```
disable /tty01
```

disables the terminal connected to serial line */dev/tty01*.

3. Turn off the power to the terminal.
4. Disconnect the terminal from the system.

7.4 Setting the Terminal Type

Several XENIX utility programs (for example the visual editor, **vi(C)**, and the visual shell, **vsh(C)**), and many "screen-oriented" application programs, must make use of detailed information about your terminal. These programs communicate with the terminal hardware to move the cursor, highlight an area of the screen, clear the screen, and the like.

XENIX sets the terminal type for you during installation. The console and available serial lines are automatically assigned default settings.

The standard XENIX shell **sh(C)** (command interpreter) sets aside a variable, **TERM**, to refer to the name of your terminal. This variable is then passed on to programs that you invoke, so your

terminal type is available to them if they need it. The file */etc/termcap* (short for “terminal capabilities”) is an ASCII database that describes features of over 100 popular terminals. A list of terminals supported by XENIX, along with their names, may be found in the XENIX *Reference* page **terminals(M)**. Also refer to the **termcap(M)** manual page.

The easiest way to set the **TERM** variable is with the **tset(C)** command. The **tset(C)** command determines the name of the line you have logged in on (e.g. *tty02*), then reads the file terminal type for that line.

The */etc/ttytype* file supplied with XENIX lists the available serial lines and their default settings. The file */etc/profile*, which is read and executed by the **sh(C)** shell every time you log in, will set your terminal type based on the entry in */etc/ttytype*. It contains lines such as:

```
eval 'tset -m dialup:?pc -m pc? -m: pc -e -s -r'
export PATH SHELL TERM TERMCAP TZ
```

If you log in on the console the following will appear:

```
Terminal type is pc
```

If you log in on one of the serial port lines the following prompt appears:

```
TERM = (unknown)
```

Type the name of the terminal you are using. **tset** will automatically set your **TERM** variable, and announce the terminal type on the screen. For example:

```
TERM = (unknown) vt100
```

```
Terminal type is vt100
```

If your response is not one of the names in **terminals(M)**, if you type the name incorrectly, or if you just press RETURN you see:

```
Terminal type unknown
```

In this case, you should log out and log back in again, then supply the correct name at the prompt.

You may modify both */etc/ttytype* and the **tset** command line in */etc/profile* to suit your particular needs. Examples for customizing terminal settings are provided on the **tset(C)** manual page in the XENIX *Reference*.

7.5 Changing Serial Line Operation

Whenever you enable a terminal with the **enable(C)** command, the system automatically sets the characteristics of the serial line such as baud rate and parity to a set of default values. Sometimes these values do not match the values used by the terminal, and therefore must be changed to allow communication between the system and the terminal. You can display and change the operating characteristics of a serial line with the **stty(C)** (for “set tty”) command.

You can display the current operating characteristics of a serial line by typing

```
stty
```

at the terminal connected to that line. If it is impossible to login at that terminal, you may use another terminal to display the characteristics. Log in as the super-user at the console and type

```
stty < specialfile
```

where *specialfile* is the name of the device special file corresponding to the serial line (see Appendix A). For example, the command

```
stty < /dev/tty01
```

displays the current characteristics of the serial line named */dev/tty01*. The command displays

the baud rate, the parity scheme, and other information about the serial line. The meaning of this information is explained in **stty(C)** in the *XENIX Reference*.

One common change to a serial line is changing the baud rate. This is usually done from a terminal connected to another serial line since changing the rate disrupts communication between the terminal and system. Log in as the super-user at the other terminal and type

```
stty baud-rate < specialfile
```

where *baud-rate* is the terminal's desired baud rate and *specialfile* is the name of the device special file corresponding to the serial line you wish to change. The baud rate must be in the set 50, 75, 110, 134, 150, 200, 300, 600, 1200, 2400, 4800, and 9600. For example, the command

```
stty 9600 < /dev/tty01
```

changes the baud rate of the serial line */dev/tty01* to 9600. Note that the "less than" symbol (<) is used for both displaying and setting the serial line from another terminal.

Another common change is changing the way the system processes input and output through the serial line. Such changes are usually made from the terminal connected to the serial line. For example, the command

```
stty -tabs
```

causes the system to expand tabs with spaces (used with terminals which do not expand tabs on their own), and the command

```
stty echoe
```

causes the system to remove a deleted character from the terminal screen when you back over it with the BACKSPACE key.

Note that the **stty** command may also be used to adapt a serial line to an unusual terminal or to another type of serial device which requires parity generation and detection, and special input and output processing.

For a full description of this command, see **stty(C)** in the *XENIX Reference*.

7.6 Setting Serial Line Baud Rate

The changes that **stty(C)** makes to any aspect of a serial line (described in the preceding section) are volatile in the sense that they disappear when the line 'closes' (usually when the current user logs out). To make a change that will carry over to the next login, follow these steps:

1. Log in as root (super-user) on the console. Make sure nobody is logged in on the line you want to change.
2. **disable** the line you want to change. See **disable(C)** for instructions. When using the **disable** or **enable(C)** commands, make sure that you wait a full minute between each use of the command. Failure to do so can cause a system crash.
3. Edit the file */etc/ttys*. The format of this file is described on the **ttys(M)** manual page. Find the entry corresponding to the serial line whose speed you want to change, then change the one character 'mode' to reflect the new speed. The **getty(M)** page furnishes a table of corresponding speeds.
4. **enable** the serial line. See **enable(C)** for instructions. The speed change will be read by **getty(M)**, and a login message will appear at the new baud rate on the port.

Chapter 8

Solving System Problems

- 8.1 Introduction 1
- 8.2 Restoring a Nonechoing Terminal 1
- 8.3 Freeing a Jammed Lineprinter 1
- 8.4 Stopping a Runaway Process 2
- 8.5 Replacing a Forgotten Password 2
- 8.6 Removing Hidden Files 3
- 8.7 Restoring Free Space 3
- 8.8 Restoring Lost System Files 3
- 8.9 Restoring an Inoperable System 3
- 8.10 Recovering from a System Crash 3
- 8.11 Changing XENIX Initialization 4
 - 8.11.1 Changing the /etc/rc File 4
 - 8.11.2 Changing the .profile Files 5
 - 8.11.3 Changing the /etc/motd File 5

8.1 Introduction

This chapter explains how to solve problems that affect the operation of the system. The problems range in complexity from how to fix a nonechoing terminal, to how to restore lost system files.

8.2 Restoring a Nonechoing Terminal

A nonechoing terminal is any terminal that does not display characters typed at the keyboard. This situation can occur whenever a program stops prematurely as a result of an error, when the user presses the BREAK key, or when the user inadvertently presses a CTRL-S.

To restore the terminal to normal operation, follow these steps:

1. Press the CTRL-Q key. This restarts transmission from the computer in case the CTRL-S key was accidentally pressed.
2. Press the RETURN key several times.

If the terminal is still not echoing, continue with the following steps.

1. Press the CTRL-J key. The system may display an error message. If it does, ignore the message.
2. Type

`stty sane`

and press the CTRL-J key. The terminal does not display what you type, so type carefully.

After pressing the CTRL-J key, the terminal should be restored and you may continue your work.

8.3 Freeing a Jammed Lineprinter

Lineprinter errors, such as running out of paper, can cause the **lpd** program to “lock up” the printing queue, preventing the current file and any other files in the queue from being printed. The **lpd** program is the “lineprinter daemon”, the program which does the actual printing for the system print command **lpr**.

To free a jammed lineprinter, follow these steps:

1. Log in as the super-user.
2. Type

`ps -a`

to find the process identification number (PID) of the **lpd** program. (The PID is in the first column of the display.) The command display should look like this:

PID	TTY	TIME	COMMAND
34	01	0:08	sh
135	01	0:25	lpd

3. Type

`kill PID`

and press the RETURN key. The *PID* is the process identification number of the program.

4. Locate and fix the error that caused the lineprinter to become jammed.

5. Type

```
cd /usr/spool/lpd
```

to change to the lineprinter spool directory. This directory temporarily holds the files to be printed.

6. Type

```
rm -f lock
```

to remove the lineprinter spool's lock file. This frees the queue and allows printing to continue.

After freeing the lineprinter, you must issue another **lpr** command to start printing.

8.4 Stopping a Runaway Process

A runaway process is a program that cannot be stopped from the terminal at which it was invoked. This occurs whenever an error in the program "locks up" the terminal, that is, prevents anything you type from reaching the system.

To stop a runaway process, follow these steps:

1. Go to a terminal that is not locked up.

2. Log in as the super-user.

3. Type

```
ps -a
```

and press the RETURN key. The system displays all current processes and their process identification numbers (PIDs). Find the PID of the runaway program.

4. Type

```
kill PID
```

and press the RETURN key. The *PID* is the process identification number of the runaway program. The program should stop in a few seconds. If the process does not stop, type

```
kill -9 PID
```

and press the RETURN key.

The last step is sure to stop the process, but may leave temporary files or a nonechoing terminal. To restore the terminal to normal operation, follow the instructions in the section "Restoring a Nonechoing Terminal" in this chapter.

8.5 Replacing a Forgotten Password

The XENIX operating system does not provide a way to decipher an existing password. If a user forgets his password, the system manager must change the password to a new one. To change an ordinary user password, follow the instructions in the section "Changing a User's Password" in

Chapter 3.

8.6 Removing Hidden Files

A hidden file is any file whose name begins with a dot (.). You can list the hidden files in a directory by typing:

```
ls -a
```

and pressing the RETURN key.

You can remove most hidden files from a directory by typing:

```
rm .[a-z]*
```

and pressing the RETURN key. Remaining files can be removed individually.

8.7 Restoring Free Space

The system displays an “out of space” message whenever the root directory has little or no space left to work. To restore system operation, you must delete one or more files from the root directory. To delete files, follow the steps outlined in the section “Maintaining Free Space” in Chapter 5.

8.8 Restoring Lost System Files

If a system program or data file is accidentally modified or removed from the file system, you can recover the file from the periodic backup disk with the *sysadmin* program. To restore the files, follow the instructions in the section “Restoring a Backup File” in Chapter 6.

8.9 Restoring an Inoperable System

On very rare occasions, one or more of the critical XENIX system files may be accidentally modified or removed, preventing the system from operating. In such a case, you must reinstall the XENIX system, and restore user program and data files from backup disks. To reinstall the system, follow the instructions in the *XENIX Installation Guide*. To restore files from backup disks, follow the instructions in the section “Restoring a Backup File” in Chapter 6.

8.10 Recovering from a System Crash

A system crash is a sudden and dramatic disruption of system operation that stops all work on the computer. System crashes occur very rarely. They are usually the result of hardware errors or damage to the root file system which the operating system cannot correct by itself. When a system crash occurs, the system usually displays a message explaining the cause of the error, then stops. This gives the system manager the chance to recover from the crash by correcting the error (if possible) and restarting the system.

A system crash has occurred if 1) the system has displayed at the system console a message beginning with “panic:”, or 2) the system refuses to process all input (including INTERRUPT and QUIT keys) from the system console and all other terminals.

To recover from a system crash, follow these steps:

1. Use the error message(s) displayed on the system console to determine the error that caused the crash. If there is no message, skip to step 3.

2. Correct the error, if possible. A complete list of error messages and descriptions for correcting the errors is given in *messages(M)* in the *XENIX Reference Manual*. (Even if the problem cannot be located or corrected, it is generally worthwhile to try to restart the system at least once by completing the remaining steps in this procedure.)
3. Press the reset button, or turn off the computer, and follow the steps described in Chapter 2, "Starting the System," to restart the system.
4. If the system will not restart, or crashes each time it is started, the operating system is inoperable and must be reinstalled. Follow the procedures described in the *XENIX Installation Guide* for installing the system from the distribution media. You may then be able to boot the system from the hard disk, then restore damaged files. Refer to the *XENIX Operations Guide* Chapter 6, "Backing Up File Systems," to restore user's files.
5. If the system cannot be started from the "Boot" disk in the distribution set for installation, the computer has a serious hardware malfunction. Contact a hardware service representative for help.

8.11 Changing XENIX Initialization

One common problem is how to adapt the system initialization to suit your system environment. This problem occurs whenever you have added new devices such as terminals or disk drives to the system, and wish these devices to be automatically enabled or mounted whenever you start normal system operation. You can adapt system initialization by modifying the system initialization files.

The XENIX initialization files contain XENIX commands and/or data which the system reads at system startup or whenever a user logs in. The files typically mount file systems, start programs, and set home directories and terminal types. The initialization files are named */etc/rc*, *.profile*, and */etc/motd*.

The system manager may modify these files to create any desired initial environment. The files are ordinary text files and may be modified using a text editor such as **ed** (see the *XENIX User's Guide*). Note, however, that the */etc/rc* and *.profile* files contain XENIX commands and comments, and have the command file format described in Chapter 7, "The Shell," in the *XENIX User's Guide*.

8.11.1 Changing the */etc/rc* File

The */etc/rc* file contains XENIX system initialization commands. The system executes the commands at system startup. The commands display a startup message, start various system daemons, and mount file systems. You can display the contents of the file with the **more** command. Type

```
more /etc/rc
```

and press the RETURN key.

The */etc/rc* file executes the file */etc/rc.user* if one exists. It is recommended that any changes to the *rc* file be done by editing */etc/rc.user*; however, */etc/rc* may be changed if necessary.

You may change the contents of the file so that the system executes any set of commands you wish. For example, if you want the system to automatically mount a new file system, simply append the appropriate **mount** command in the file. The system will execute the command on each startup.

To append a command to the file, follow these steps:

1. Log in as the super-user.
2. Invoke a text editor and specify the */etc/rc.user* as the file to be edited.
3. Locate the place in the file you wish to insert the command (e.g., if the command mounts a file system, insert it with other mounting commands).
4. Insert the command on a new line. Make sure you type the command correctly. The system will reject any incorrect command and all following commands when it reads the file at system startup.
5. Exit the editor.

No other changes to the file are required. Be careful not to delete any commands already in the file unless you are sure they are not needed.

8.11.2 Changing the *.profile* Files

The *.profile* files contain commands that initialize the environment for each user. The commands in the file are executed whenever the user logs in. The file usually contains commands that set and export various system variables (e.g., TERM, PATH, MAIL). These variables give the system information such as what terminal type is being used, where to look for programs the user runs, where to look for the user's mailbox, what keys to expect for the "kill" and "backspace" functions, and so on (see Chapter 7, "The Shell," in the *XENIX User's Guide*).

There is one *.profile* file for each user account on the system. The files are placed in the user's home directory when the account is created. An ordinary user may modify his own *.profile* file, or allow the system manager to make modifications. In either case, the file can be edited like the */etc/rc* file, using a text editor. Commands can be added or removed as desired.

8.11.3 Changing the */etc/motd* File

The message of the day file, */etc/motd*, contains the greeting displayed whenever a user logs in. Initially, this file contains the name and version number of the XENIX system. It can be modified to include such messages as a reminder to clean up directories, a notice of the next periodic backup, and so on.

The */etc/motd* file is an ordinary text file, so you can change the message by editing the file with a text editor. One common change is to include a reminder to delete unused files in order to preserve disk space. In general, you should limit the size of the file to include no more than a screenful of information.

Chapter 9

Building a Micnet Network

9.1 Introduction	1
9.2 Planning a Network	1
9.2.1 Choosing Machine Names	1
9.2.2 Choosing a Network Topology	2
9.2.3 Drawing a Network Topology Map	2
9.2.4 Assigning Lines and Speeds	2
9.2.5 Choosing Aliases	3
9.3 Building a Network	4
9.3.1 Creating the Micnet Files	4
9.3.2 Saving the Micnet Files	7
9.3.3 Restoring Micnet Files	7
9.4 Starting the Network	8
9.5 Testing a Micnet Network	9
9.5.1 Checking the Network Connections	9
9.5.2 Using the LOG File to Locate a Problem	9
9.5.3 Stopping the Network	10
9.5.4 Modifying the Micnet Network	11
9.6 Using a Uucp System	11

9.1 Introduction

A Micnet network allows communications between two or more independent XENIX systems. The network consists of computers connected by serial communication lines (that is, RS-232 ports connected by cable). Each computer in the network runs as an independent system, but allows users to communicate with the other computers in the network through the **mail**, **rcp**, and **remote** commands. These commands pass information such as mail, files, and even other commands from one computer to another.

It is the system manager's task to build and maintain a Micnet network. The system manager decides how the computers are to be connected, makes the actual physical connections, then uses the **netutil** program to define and start the network.

This chapter explains how to plan a network and then build it with the **netutil** program. In particular, it describes:

- How to choose machine names and aliases
- How to draw the network topology map
- How to assign serial lines
- How to create the Micnet files
- How to distribute the Micnet files
- How to test the Micnet network

9.2 Planning a Network

To build a Micnet network, the **netutil** program will require you to provide the names of the computers that will be in the network, a description of how the computers are to be connected, a list of the serial lines to be used, the names of the users who will use the network, and what aliases (if any) they will be known by.

To keep the task as simple as possible, you should take some time to plan the network, and make lists of the information you will be required to supply. To help you make these lists, the following sections suggest ways to plan a network.

9.2.1 Choosing Machine Names

A Micnet network requires that each computer in the network have a unique "machine name." A machine name helps distinguish each computer from other computers in the network. It is best to choose machine names as the first step in planning the network. This prevents confusion later on when you build the network with the **netutil** program.

A machine name should suggest the location of the computer or the people who are users on the computer; however, you may use any name you wish. The name must be unique and consist of letters and digits. The Micnet programs use only the first eight characters of each name, so be sure those characters are unique.

The **netutil** program saves the machine name of a computer in a */etc/systemid* file. One file is created for each computer. After you have built and installed the network, you can find out the machine name of the computer you are using by displaying the contents of this file.

9.2.2 Choosing a Network Topology

The network topology is a description of how the computers in the network are connected. In any Micnet network, there are two general topologies from which all topologies can be constructed. These are “star” and “serial”.

In a star topology, all computers are directly connected to a central computer. All communications pass through the central computer to the desired destination.

In a serial topology, the computers form a chain, with each computer directly connected to no more than two others. All communications pass down the chain to the desired destination.

A network may be strictly star, strictly serial, or a combination of star and serial topologies. The only restriction is that no network may form a ring. For example, you cannot close up a serial network by connecting the two computers at each end.

The kind of topology you choose depends on the number of computers you have to connect, how quickly you want communications to proceed, and how you want to distribute the task of passing along communications. A star topology provides fast communication between computers, but requires both a large portion of the central computer's total operation time, and a large number of serial lines on the central computer. A serial topology distributes the communication burden evenly, requiring only two serial lines per computer, but it is slow if the chain is very long (communication between computers can take several minutes). Often a combination of star and serial topologies makes the best network. In any case, make the choice you think best. If you discover you have made a wrong choice, you may change the network at any time.

9.2.3 Drawing a Network Topology Map

A network topology map is a sketch of the connections between computers in the network. You use the map to plan the number and location of the serial lines used to make the network.

You can make the map while you work out the topology. Simply arrange the machine names of each computer in the network on paper, then mark each pair of computers you wish to connect with serial lines. For example, the topology map for three computers might look like this:

a ----- b ----- c

As you draw, make sure that there is no more than one connection between any two computers in the network. Furthermore, make sure that no rings are formed (a ring is a series of connections that form a closed circle). Multiple connections and rings are not permitted.

9.2.4 Assigning Lines and Speeds

Once you have made the topology map, you can decide which serial lines to use. Since every connection between computers in the network requires exactly two serial lines (one on each computer), you need to be very careful about assigning the lines. Follow these steps:

1. Make a list of the serial lines (tty lines) available for use on each computer in the network. You can display a list of the serial lines on a computer by displaying the file */etc/ttys*. A line is available if it is not connected to any device such as a terminal or modem.
2. Using the topology map, first pick a computer, then assign one, and only one, serial line to each connection shown for that computer. The serial lines must be from the list of available lines for that computer. No line may be assigned more than once. For example, if computer “a” has only one available serial line (tty01), then the topology

map should look like this:

```
a ----- b ----- c
tty01
```

3. Repeat step 2 for all computers in the topology map. Make sure that each connection is assigned a line and that no two connections on any given computer have the same line. When finished, the map should look like this:

```
a ----- b ----- c
tty01  tty02  tty03  tty04
```

If a computer does not have enough available serial lines to meet its needs, you can make the lines available by removing the devices already connected to them. If you cannot remove devices you must redraw your topology map.

4. Using the topology map, assign a serial line transmission speed for each computer pair. The speed may be any in the normal range for XENIX serial lines (typically 110 to 9600). Transmission speeds are a matter of preference. In general, a higher speed means a smaller amount of time to complete a transmission, but a greater demand on system's input and output capabilities. In some case, transmission speeds are a matter of hardware capabilities. Some hardware is not capable of transmission speeds greater than 1200 baud. For this reason, 1200 is the recommended speed when first installing Micnet. You may then increase the speed if you find the hardware can support it.
5. After the topology map is completely filled in, make a list of all computer pairs, showing their machine names, serial lines, and transmission speeds. You will use this list when installing the network.

9.2.5 Choosing Aliases

Once you have decided how to connect the computers in the network, you can choose aliases for users in the network. An alias is a simple name that represents both a location (computer) and a user. Aliases are used by the **mail** command to allow you to refer to specific computers and users in a network without giving the explicit machine and user names. Although not a required part of the network, aliases can make the network easier to use and maintain.

There are three kinds of aliases: standard, machine, and forward. A standard alias is a name for a single user or a group of users. A machine alias is a name for a computer or an entire network (called a site). A forward alias is a temporary alias for a single user or group of users. A forward alias allows users who normally receive network communications at one computer to receive them at another.

When you build a network with the **netutil** program, you will be asked to provide standard aliases only. (You can incorporate machine and forward aliases into the network at your leisure.) Each standard alias must have a unique name and a list of the login names of the users it represents. You may choose any name you wish as long as it consists of letters and numbers, begins with a letter, and does not have the same spelling as the login names. The name should suggest the user or group of users it represents. The login names must be the valid login names of users in the network.

To help you prepare the aliases for entry during the **netutil** program, follow these steps:

1. Make a list of the user aliases (i.e., the aliases that refer to just one user) and the corresponding login names of each user.
2. Make a separate list of the group aliases (i.e., the aliases that refer to two or more users) and the login names or user aliases (from the first list) of the corresponding users. A group alias may have any number of corresponding users.

Note that there are a number of predefined group aliases. The name **all** is the predefined alias for all users in the network. The *machine-names* of the computers in the network are predefined aliases for the users on each computer. Do not use these names when defining your own aliases.

9.3 Building a Network

You build a network with the **netutil** program. The program allows you to define the machines, users, and serial lines that make up the network.

To build a network, you must first create the Micnet files that define the network, then transfer these files to each computer in the network. After each computer receives the files, you may start the network and use it to communicate between computers.

The following sections describe how to build the network.

9.3.1 Creating the Micnet Files

The Micnet files are created with the **install** option of the **netutil** program. The **install** option asks for the names, aliases, and serial lines of each computer in the network. As you supply the information, it automatically creates the files needed for each computer. These files can then be transferred to the other computers in the network with the **save** and **restore** options of **netutil**. This means you can build the entire network from just one computer.

To use the **install** option, follow these steps:

1. Log in as the super-user.
2. Type
 netutil
and press the RETURN key. The program displays the network utility menu. The **install** option is the first item in the menu.
3. Type the number *1* and press the RETURN key. The program displays the following message.
 Compiling new network topology
 Overwrite existing network files? (yes/no)?

Type *y* and press the RETURN key to overwrite the files. The existing network files must be overwritten to create the new network. The first time you install the network, these files contain default information that need not be saved. If you install the system a second time or expand the system, it may be wise to save a copy of these files before starting the **install** option. The files can be saved on a floppy disk with the **save** option described later in this chapter.

Once you have typed *y* the program displays the following message.

Enter the name of each machine
(or press RETURN to continue installation).
Machine name:

4. Enter a machine name by typing the name and pressing the RETURN key. You may enter more than one name on a line by separating each with a comma or a space. After you have entered all the names, simply press the RETURN key to continue to the next step. The program displays the names you have entered and asks if you wish to make changes.
5. Type *y* (for "yes") if you wish to enter all the names again. Otherwise, type *n* (for "no") or just press the RETURN key to move on to the next step. If you type *n*, the program displays the message:

For each machine, enter the names of the machines
to be connected with it
Machine a:
Connect to:

6. Using the list of machine pairs you created when planning the network, enter the machine names of the computers connected to the given computer. You may enter more than one name on a line by separating each name with a comma (,) or a space. When you have entered the machine names of all computers connected to the given computer, press the RETURN key. The program asks for the names of the computers connected to the next computer.
7. Repeat step 5 for all remaining computers. As the program asks for each new set of connections, it will show a list of the machine names it already knows to be connected with the current computer. You need not enter these names. The program automatically checks for loops. If it finds one, it ignores the machine name that creates the loop and asks for another.

Finally, when you have given the connections for all computers in the network, the program displays a list of the connections and asks if you wish to make corrections.

8. Type *y* if you wish to enter the connections again. Otherwise, type *n* to move to the next step. If you type *n*, the program displays the message:

For each machine pair, enter the tty name and tty speeds
For the a <==> b machine pair.
Tty on a:

9. Using the list of serial line assignments you created when planning the network, type the serial line name or number (e.g., tty03 or 3) for the first computer in the pair and press the RETURN key. The program displays the message:

Tty on b:

10. Type the serial line name for the second computer in the pair and press the RETURN key. The program displays the message:

Speed:

11. Type the speed (e.g., 1200) and press the RETURN key. The program asks for the serial lines and transmission speed of the next pair.

12. Repeat step 7 for all remaining machine pairs. When you have given serial lines and speeds for all pairs, the program displays this information and asks if you wish to make corrections.

13. Type *y* if you wish to enter the serial lines and speeds again. Otherwise, type *n* to move to the next step. The program displays the message:

Enter the names of users on each machine:

For machine a:

Users on a:

14. Enter a name by typing the login name of a user on the given computer, then press the RETURN key. You may enter more than one name on a line by separating each name with a comma (,) or a space. When you have entered all names for the given computer, press the RETURN key. The program displays the names of the users on the computer and asks if you wish to make corrections.
15. Type *y* if you wish to enter the user names again. Otherwise, type *n*. If you type *n*, the program asks for the users on the next computer.

16. Repeat step 10 and 11 for all remaining computers. When you have given names of users for every computer, the program asks if you wish to enter aliases.

Do you wish to enter any aliases? (yes/no)?

17. Type *y* if you wish to enter aliases. Otherwise, type *n* to complete the installation. If you type *y*, the program displays the message:

Each alias consists of two parts, the first is the alias name,
the second is a list of one or more of the following:

valid user names
previously defined aliases
machine names

Aliases:

18. Using the list of aliases you created when planning the network, type the name of an alias and press the RETURN key. The program displays the message:

Users/Aliases:

19. If the alias is to name a single user, type the login name of that user and press the RETURN key. The program asks for another alias.

If, on the other hand, the alias is to name several users, type the login names of the users. If one or more of the users to be named by the alias are already named by other aliases, type the aliases instead of the login names. If all the users on one computer are to be named by the alias, type the machine name instead of the login names. In any case, make sure that each item typed on the line is separated from the next by a comma (,) or a space. If there are more items than can fit on the line, type a comma after the last item on that line and press the RETURN key. You can then continue on the next line. After all names and aliases have been typed, press the RETURN key. The program asks for another alias.

20. Repeat steps 12 and 13 for all remaining user aliases in your list. When you have given all aliases, press the RETURN key. The program displays a list of all aliases and their users and asks if you wish to make corrections.
21. Type *y* if you wish to enter all aliases again. Otherwise, type *n* to complete the installation.

Once you direct **netutil** to complete the installation, it copies the information you have supplied to the network files, displaying the name of each file as it is updated. Once the files are updated, you may use the **save** option to copy the Micnet files to floppy disk.

9.3.2 Saving the Micnet Files

You can save copies of the Micnet files on floppy disk with the **save** option of the **netutil** program. Saving the files allows you to transfer them to the other computers in the network. Before you can save the files, you need to format a floppy disk (see the section, "Formatting Floppy Disks," in Chapter 4).

To save the files, follow these steps:

1. Log in as the super-user.
2. Type
`netutil`
and press the RETURN key. The program displays the network utility menu.
3. Type the number 2 and press the RETURN key. The program displays the message:
`Save to /dev/fdx (yes/no)?`
where *x* is a drive number.
4. If you wish to use the specified disk drive, insert a blank, formatted floppy disk into the drive, wait for the drive to accept the disk, then type *yes* and press the RETURN key. If you do not wish to use the drive, type *no* and press the RETURN key. The program displays a prompt asking you for the filename of the disk drive you wish to use. Insert a blank, formatted disk into your chosen disk, wait for the drive to accept the disk, then type the filename of the drive.

In either case, the program copies the Micnet files to the floppy disk.
5. Remove the floppy disk from the drive. Using a soft tip marker (do not use a ball point pen), label the disk "Micnet disk".

As soon as all files have been copied, you can transfer them to all computers in the network.

9.3.3 Restoring Micnet Files

The last step in building a Micnet network is to copy the Micnet files from the Micnet disk to all computers in the network. Do this with the **restore** option of the **netutil** program. For each computer in the network, follow these steps:

1. Log in as the super-user.
2. Type:
`netutil`
and press the RETURN key. The program displays the network utility menu.
3. Type the number 3 and press the RETURN key. The program displays the message:
`Restore from /dev/fdx (yes/no)?`
where *x* is the number of a drive.

4. If you wish to use the specified disk drive, insert the micnet disk into the drive, wait for the drive to accept the disk, then type *yes* and press the RETURN key. If you do not wish to use the drive, type *no* and press the RETURN key. The program displays a prompt asking you for the filename of the disk drive you wish to use. Insert the micnet disk into your chosen drive, wait for the drive to accept the disk, then type the filename of the drive.

In either case, the program copies the network files to the appropriate directories, displaying the name of each file as it is copied. Finally, the program displays the message:

Enter the name of this machine:

5. Type the machine name of the computer you are using, and press the RETURN key. The program copies this name to the new */etc/systemid* file for the computer. If necessary, it also disables the serial lines to be used on the computer, preparing them for use with the network.

When the files have been copied, you may start the network with the **start** option.

9.4 Starting the Network

Once the Micnet files have been transferred to a computer, you can start the network with the **start** option of the **netutil** program. The **start** option starts the Micnet programs which perform the tasks needed to communicate between the computers in the network.

To start the network, follow these steps for each computer in the network:

1. Log in as the super-user.
2. Type:

```
netutil
```


and press the RETURN key. The system displays the network utility menu.
3. Type *4* and press the RETURN key. The program searches for the */etc/systemid* file. If it finds the file, it starts the network. If not, it asks you to enter the machine-name of the computer and then creates the file. The program also asks if you wish to log errors and transmissions. In general, these are not required except when checking or testing the network. When starting the network for the first time, type *n* to each question and press the RETURN key.

Once the network has started, you may move to the next computer and start the network there.

Note that, for convenience, you can let each computer start the network automatically whenever the system itself is started. Simply include the command

```
netutil start
```

in the system initialization file */etc/rc* of each computer. To add this command, use a text editor as described in the section, "Changing the */etc/rc* File," in Chapter 8. You can add the *-x* or *-e* options to this command line if you wish to log transmissions or errors. Even if you do not use these options, Micnet copies a log in and log out message to the system *LOG* file each time you start and stop the network. This means you will need to periodically clear the file. See the section, "Clearing Log Files," in Chapter 5.

9.5 Testing a Micnet Network

After you have started a network for the first time, you should test the network to see that it is properly installed. In particular, you must determine whether or not each computer is connected to the network.

To test the network, you will need to know how to use the **mail** command (see Chapter 6, “Mail,” in the *XENIX User's Guide*). The following sections explain how to test the network and how to correct the network if problems are discovered.

9.5.1 Checking the Network Connections

You can make sure that all computers are connected to the network by mailing a short message to **all** (the alias for all users in the network) with the **mail** command. Follow these steps:

1. Choose a computer.
2. Log in as the super-user.
3. Use the **mail** command (see the *XENIX User's Guide*) and the **all** alias to mail the message:

Micnet test

to all users in the network.

4. Check the mailboxes of each user in the network to see if the message was received. To check the mailboxes, log in as the super-user at each computer and use the **cat** command to display the contents of each user's mailbox. The name of each user's mailbox has the form:

/usr/spool/mail/*login-name*

where *login-name* is the user's login name.

If all users have received the message, the network is properly installed. If the users at one or more computers fail to receive the message, the computers are not properly connected to the network. To fix the problem, you need to locate the computer which has failed to make a connection. The next section explains how to do this.

9.5.2 Using the LOG File to Locate a Problem

You can locate a problem with connections by examining the *LOG* files on each computer in the network. The *LOG* files contain a record of the interaction between each pair of computers. There are two *LOG* files for each pair of computers (one file on each computer). The *LOG* files on any given computer are kept in subdirectories of the */usr/spool/micnet* directory. Each subdirectory has as its name the *machine-name* of the other computer in the pair. You can examine the contents of a *LOG* file by typing

cat /usr/spool/micnet/remote/*machine-name*/LOG

and pressing the RETURN key. The *machine-name* must be the name of a computer that is paired with the computer you are using.

Each *LOG* file should contain a “startup message” which lists the name of each computer in the pair and the serial line through which the pair is connected. It also shows the date and time at which the network was started. The message should look like:

XENIX Operations Guide

```
daemon.mn: running as MASTER
Local system: a
Remote system: b, /dev/tty02
Tue Sep 27 22:30:35 1983
```

A startup message is added to the file each time the network starts successfully. If the message is not present, then one or more of the the network files and directories cannot be found. Make sure that you have used the **restore** option to transfer all the network files to the computer. Also, make sure that the */etc/systemid* file contains the correct machine name for the given computer.

Each *LOG* file will contain a “handshake” message if the connection between the computer pair has been established. The message

```
first handshake complete
```

is added to the file on a successful connection. If the message is not present, make sure that the network has been started on the other computer in the pair. The network must be started on both computers before any connection can be made. If the network is started on both computers yet no handshake message appears, then the serial line may be damaged or improperly connected. Check the serial line to make sure that the cable is firmly seated and attached to the correct RS-232 connectors on both computers. If necessary, replace the cable with one known to work.

If both the startup and handshake messages appear in the *LOG* file but the network is still not working, then there is a problem in transmission. You can create a record of the transmissions and errors encountered while transmitting by restarting the network and requesting Micnet to log all transmissions and errors. Just type *y* (for “yes”) when the **start** option asks if you wish to log errors or transmissions.

Error entries contain the error messages generated during transmission. Each message lists the cause of the error and the subroutine which detected the error. For example, the message

```
rsync: bad Probe resp: 68
```

shows that the *rsync* subroutine received a bad response (character 68 hexadecimal) from the other computer. You may use this information to track down the cause of the problem. One common problem is stray information being passed down the serial line by electronic noise. Make sure that the serial line’s cable is properly protected against noise, e.g., make sure it does not lie near any electric motor, generator, or other source of electromagnetic radiation. Also, make sure the cable is in good condition.

Transmission entries contain a record of normal transmissions between computers. Each entry lists the direction, byte count, elapsed time, and time of day of the transmission. For example, the entry

```
rx: 0c 01 22:33:49
```

shows that 12 characters (0c hexadecimal) were received (*rx*) at 22:33:49. The elapsed time for the transmission was 1 second. You can use the records to see if messages are actually being transmitted.

9.5.3 Stopping the Network

You can stop the network with the **stop** option of the **netutil** program. This option stops the Micnet programs, stopping communication between computers in the network.

To stop the network, follow these steps on each computer in the network:

1. Log in as the super-user.
2. Type

`netutil`

and press the RETURN key. The program displays the network utility menu.

3. Type 5 and press the RETURN key. The program stops the network programs running on the computer.

9.5.4 Modifying the Micnet Network

You can modify a Micnet network at any time by changing one or more of the Micnet files. You can reinstall the network with the `netutil` program. For very small changes (for example, correcting the spelling of an alias), you can modify the Micnet files directly with a text editor. The files and their contents are described in detail in the M section of the *XENIX Reference Manual*.

Before making any changes to a file, a copy should be made. You can make a copy with the `cp` command. You can replace an old file with the updated file using the `mv` command. Once one or more files have been changed on one computer, the files must be transferred to the other systems in the network using the `save` and `restore` options. These options can only be used after you have stopped the network.

Note that changes to the *aliases* file will not be incorporated into the system until the `aliashash` program is executed. This program produces the *aliases.hash* file needed by the network to resolve aliases. See *aliashash(M)* in the *XENIX Reference Manual* for a description of this command.

9.6 Using a Uucp System

You can send and receive mail from other Micnet sites by installing a uucp system on one computer in your site. A uucp system is a set of XENIX programs that provide communication between computers using ordinary telephone lines.

To use a uucp system with your Micnet network, follow these steps:

1. Install a uucp system on one computer in the Micnet site. Installation of a uucp system requires a modem and the uucp software provided with the *XENIX Software Development System*. See the *XENIX Programmer's Guide* for complete details.
2. Add the entry
`uucp:`
to the *aliases* file of the computer on which the uucp system is installed.
3. For all other computers in your site, add the entry
`uucp:machine-name:`
to the *aliases* file. The *machine-name* must be the name of the computer on which the uucp system is installed. One may also use the longer form of entry on the computer on which the uucp system is installed.

You can test the uucp system by mailing a short letter to yourself via another site. For example, if you are on the site "chicago", and there is another Micnet site named "seattle" in the system, then the command

`mail seattle!chicago!johnd`

will send mail to the "seattle" site, then back to your "chicago" site, and finally to the user "johnd" in your Micnet network. Note that a uucp system usually performs its communication

tasks according to a fixed schedule, and may not return mail immediately.

Appendix A

XENIX Special Device Files

A.1 Introduction A-1

A.2 File System Requirements A-1

A.3 Special Filenames A-1

A.4 Block Sizes A-1

A.5 Gap and Block Numbers A-2

A.6 Terminal and Network Requirements A-2

A.1 Introduction

This appendix contains information needed to create file systems and add terminals to the XENIX system. For a full description of the special files mentioned here, see the XENIX *Reference Manual*.

A.2 File System Requirements

Many of the file system maintenance tasks described in this guide require the use of special filenames, block sizes, and gap and block numbers. The following sections describe each in detail.

A.3 Special Filenames

A special filename is the name of the device special block or character I/O file corresponding to a peripheral device, such as a hard or floppy disk drive. These names are required in such commands as **mkfs**, **mount**, and **df** to specify the device containing the file system to be created, mounted, or searched. The following table lists the special filenames and corresponding devices that you may use for your computer. See Appendix B "XENIX Directories" for a list of the other special files in the */dev* directory.

Block I/O	
Special Filename	Disk Drive
<i>/dev/fd0</i>	Floppy Drive 0
<i>/dev/fd1</i>	Floppy Drive 1
<i>/dev/fd2</i>	Floppy Drive 2
<i>/dev/fd3</i>	Floppy Drive 3
<i>/dev/hd0</i>	Hard Disk Drive 0
<i>/dev/hd1</i>	Hard Disk Drive 1
<i>/dev/hd2</i>	Hard Disk Drive 2
<i>/dev/hd3</i>	Hard Disk Drive 3
<i>/dev/cd0</i>	Cartridge Drive 0
<i>/dev/cd1</i>	Cartridge Drive 1
<i>/dev/root</i>	Root File Structure

A.4 Block Sizes

The block size of a disk is the number of blocks of storage space available on the disk, where a block is 1024 bytes of storage. The **mkfs**, and **quot** commands use block size when creating or reporting the status of a file system. Some commands traditionally report block size in 512 byte blocks. These are **df**, **du**, **ls**, **lc**, and **find**. A 500 byte file on a 1024 byte block file system is reported as using 2 blocks by these utilities, as the file uses one system block which is equivalent to 2 512 byte blocks. The size of a 10 megabyte hard disk in 1024 byte blocks is 9792. Note that some of the blocks on the disk are reserved for system use and cannot be accessed by user programs.

A.5 Gap and Block Numbers

The gap and block numbers are used by the **mkfs** command to describe how the blocks are to be arranged on a disk. The gap number for the hard disk is 3, and the blocking factor is 34.

A.6 Terminal and Network Requirements

The **enable** and **disable** commands used to add and remove serial lines from a system and the **install** option of the *netutil* program used to build a network require the names of the serial lines through which a terminal or network is to be connected. The names of the two serial lines available on your computer are */dev/tty01* and */dev/tty02*. The character I/O files corresponding to these serial lines can be found in the */dev* directory. Note that the file */dev/console*, represents a “hardwired” device and is not available for connection to terminals.

Appendix B

XENIX Directories

- B.1 Introduction B-1
- B.2 The Root Directory B-1
- B.3 The */bin* Directory B-1
- B.4 The */dev* Directory B-1
- B.5 The */etc* Directory B-2
- B.6 The */lib* Directory B-2
- B.7 The */mnt* Directory B-2
- B.8 The */tmp* Directory B-2
- B.9 The */usr* Directory B-3
- B.10 Log Files B-3

B.1 Introduction

This appendix lists the most frequently used files and directories in the XENIX system. Many of these files and directories are required for proper XENIX operation and must not be removed or modified. The following sections briefly describe each directory.

B.2 The Root Directory

The root directory (/) contains the following system directories:

/bin	XENIX command directory
/dev	Device special directory
/etc	Additional program and data file directory
/lib	C program library directory
/mnt	Mount directory (reserved for mounted file systems)
/usr	User home directories
/tmp	Temporary directory (reserved for temporary files created by programs)

All of the above directories are required for system operation.

The root directory also contains a few ordinary files. Of these files, the most notable is the *xenix* file which contains the xenix kernel image.

B.3 The /bin Directory

The /bin directory contains the most common XENIX commands, that is, the commands likely to be used by anyone on the system. The following is a list of a few of the commands.

basename	echo	passwd	su	
cp	expr	rm	sync	
date	fsck	sh	tar	
dump	login	sleep	restor	
dumpdir		mv	stty	test

These commands and all others in the /bin directory are required.

B.4 The /dev Directory

The /dev directory contains special device files which control access to peripheral devices. All files in this directory are required and must not be removed. The following is a list of some of the files.

/dev/console	System console
/dev/lp	Lineprinter
/dev/mem	Physical memory
/dev/null	Null device (used to redirect unwanted output)
/dev/rXX	Unbuffered interface to corresponding device name
/dev/root	Root file structure
/dev/swap	Swap area
/dev/ttyXX	Terminals
/dev/tty	The terminal you are using

B.5 The */etc* Directory

The */etc* directory contains miscellaneous system program and data files. All files are required, but many may be modified.

The following program and data files must not be removed or modified.

/etc/mtab	Mounted device table
/etc/mount	For mounting a file structure
/etc/mkfs	For creating a file structure
/etc/init	First process after boot

The following data files may be modified, if desired. No file may be removed.

/etc/passwd	Password file
/etc/rc	Bootup shell script
/etc/ttys	Terminal set up
/etc/termcap	Terminal capability map
/etc/motd	Message of the day

B.6 The */lib* Directory

The */lib* directory contains runtime library files for C and other language programs. The directory is required.

B.7 The */mnt* Directory

The */mnt* directory is an empty directory reserved for mounting removable file systems.

B.8 The */tmp* Directory

The */tmp* directory contains temporary files created by XENIX programs. The files are normally present when the corresponding program is running, but may also be left in the directory if the program is prematurely stopped. You may remove any temporary file that does not belong to a running program.

B.9 The */usr* Directory

The */usr* directory contains the home directories of all users on the system. It also contains several other directories which provide additional XENIX commands and data files.

The */usr/bin* directory contains more XENIX commands. These commands are less frequently used or considered nonessential to XENIX system operation.

The */usr/include* directory contains header files for compiling C programs.

The */usr/lib* directory contains more libraries and data files used by various XENIX commands.

The */usr/spool* directory contains various directories for storing files to be printed, mailed, or passed through networks.

The */usr/tmp* directory contains more temporary files.

The */usr/adm* directory contains data files associated with system administration and accounting. In particular, the */usr/adm/messages* file contains a record of all error messages sent to the system console. This file is especially useful for locating hardware problems. For example, an unusual number of disk errors on a drive indicates a defective or misaligned drive. Since messages in the file can accumulate rapidly, the file must be deleted periodically.

B.10 Log Files

A variety of directories contain log files that grow in size during the normal course of system operation. Many of these files must be periodically cleared to prevent them from taking up valuable disk space (see the section, "Clearing Log Files," in Chapter 5). The following table lists the files (by full pathname) and their contents.

Filename	Description
/etc/ddate	Records date of each backup.
/usr/adm/pacct	Records accounting information; grows rapidly when process accounting is on.
/usr/adm/messages	Records error messages generated by the system when started.
/usr/adm/wtmp	Records user logins and logouts.
/usr/adm/sulog	Records each use of the su command; grows only if option is set in the <i>/etc/default/su</i> file.
/usr/spool/at/past	Records each use of the at command.
/usr/spool/micnet/*/LOG	Records transmissions between machines in a Micnet network. The * must be the name of a remote machine connected to the current machine.

Index

B

BACKSPACE key 1-2
Backup system See File system
/bin directory contents B-1
Block
 arrangement on disk A-2
 defined 5-1
 number A-2
 ownership 5-2
 size A-1
Bootstrap program 2-1
BREAK key 1-2

C

C program
 compilation header files B-3
 library files B-2
chmod command
 permissions change 4-6
 special file protection 4-9
CNTRL-\ key 1-2
CNTRL-H key 1-2
CNTRL-Q key 1-2
CNTRL-S key 1-2
CNTRL-U key 1-2
Command
 /etc/rc file, inclusion 8-4
 location
 /bin directory B-1
 /usr/bin directory B-3
 .profile file, inclusion 8-5
 Micnet network
 connection test 9-9
 machine name
Copying floppy disks 4-9
 with dd 4-9
Copying
 directories to floppy disk 6-4
 files to floppy disk 6-4
core file, described 5-3
crypt command, file encryption 4-8

D

Daily backup See File system
/dev directory contents B-1
/dev directory
 serial line correspondence A-2
 special file 4-9
df command
 block size A-1
 free space display 5-1
 special filename A-1
Directories

Directories (*continued*)
 making copies on floppy disk 6-4
Directory
 access permissions See Permissions

 block usage 5-2
 location 5-3
 permissions See Permissions
 removal 3-8
disable command
 serial line A-2
 terminal disabling 7-2
Disk
 block number A-2
 block size A-1
 damage See File system
 free space See File system
 gap number A-2
 security 4-8
 usage 5-2
diskutil 4-3
diskutil 4-9
du command 5-2

E

enable command
 serial line A-2
 terminal enabling 7-2
Error messages B-3
ESCAPE key 1-2
/etc directory contents B-2
/etc/group file, modification 3-5
/etc/motd file
 contents 8-5
 contents B-2
 free space reminder 5-1
 modification 8-4
 modification 8-5
/etc/passwd file
 user entry 3-3
 user ID change 3-7
/etc/password file
 contents B-2
/etc/rc file
 contents 8-4
 contents B-2
 Micnet network startup 9-8
 modification 8-4
/etc/rc.user 8-4
/etc/systemid file
 machine name contents 9-1
 Micnet network startup 9-8
/etc/termcap file
 contents B-2
/etc/termcap
 termcap capabilities 7-2

Execute permission 4-5

F

File system

- amount of free space 5-1
- automatic check 5-5
- backups 6-1
 - creation 6-2
 - daily 6-1
 - disk storage 6-1
 - floppy disk labeling 6-2
 - frequency 6-1
 - listing procedure 6-2
 - periodic 6-1
 - restoration 6-3
 - schedule 6-1
 - sysadmin program 6-1
 - tar command 6-1
- cleaning 2-1
- copies 6-1
- creation 4-1
- damage
 - causes 5-4
 - restoration 5-4
- data loss 5-4
- defined 4-1
- destruction 4-1
- display free space 5-1
- expansion 5-4
- free space
 - restoration 8-3
- lack of free space 5-1
- maintaining free space 5-1
- maintenance 5-1
- mounting
 - automatic 4-3
 - initialization files 8-4
 - procedure 4-1
 - procedure 4-2
- repair 5-4
- root 4-1
- unmounting 4-1
- unmounting 4-3

File

- access
 - permissions See Permissions
- backup See File system
- core 5-3
- core file See Core file
- damage See File system
- data loss 5-4
- determining block size 5-2
- encryption 4-8
- hidden file removal 8-3
- inaccessibility 5-4
- initialization file See Initialization file
- location 5-3
- Log 5-4
- Log clearing 5-4

File (*continued*)

- lost file restoration See File system
- permissions See Permissions
- recovery from backup See File system
- removal
 - unused files 5-1
- repair See File system
- restoration See File System
- security 4-8
- system See File system
- temporary 5-3
- temporary file See temporary file
- time of last access 5-3
- unused file removal 5-1

Filename

- special filename A-1

Files

- making copies on floppy disk 6-4

find command 5-3

Floppy 4-9

Floppy disk

- block size A-1
- damage See File system
- file system creation 4-1
- Micnet file saving 9-7
- security 4-8

Floppy disks

- copying 4-9
- Formatting 4-3

Formatting floppy disks 4-3

Free space See File system

fsck command 5-4

G

Gap number A-2

Group

- access 3-6
- changing the ID 3-6
- creation 3-5
- defined 3-5
- ID 3-2
- ID 3-5
- name 3-2
- name 3-5
- number 3-2

permissions See Permissions

Group (*continued*)**H**

haltsys command 2-3
 Hard disk
 block number A-2
 block size A-1
 damage See File system
 gap number A-2
 Hidden file removal 8-3
 Home directory
 removal 3-8
 setting, initialization files 8-4
 user account 3-1

I

Initialization file
 contents 8-4
 /etc/motd file See /etc/motd file

 /etc/rc file See /etc/rc file
 modification 8-4
 .profile See .profile file
 INTERRUPT key 1-2

K

Keyboard, described 1-1
 kill command
 lineprinter freeing 8-1
 runaway process stopping 8-2
 KILL key 1-2
 l command
 listing permissions 4-5

L

/lib directory contents B-2
 Lineprinter
 jammed lineprinter 8-1
 lock file removal 8-2
 Lines, terminal connection 7-1
 LOG file
 contents 9-9
 Micnet network
 connection error location 9-9
 Log files 5-4
 Log in group 3-6
 Log in name
 Micnet network
 entry 9-6
 new user 3-2
 sending mail 5-3

Login name
 user account 3-1
 login
 terminal display 7-2
 terminal type 7-3
 lpr command 8-2
 lpr program error, lineprinter jam 8-1

M

mail command, message 5-3
 mail command
 Micnet network
 alias 9-3
 testing 9-9
 Mailbox removal 3-9
 Mail
 network See Micnet network
 /usr/spool directory B-3
 Message of the day file See /etc/motd file
 Message
 system wide message 5-2
 Micnet network
 alias
 description 9-3
 entry 9-6
 preparation 9-3
 composition 9-1
 computer
 connection test 9-9
 machine name 9-1
 connection See computer
 /etc/systemid file
 machine name contents 9-1
 system startup 9-8
 file
 copying to computers 9-7
 creation 9-4
 modification 9-11
 restoration 9-7
 saving 9-7
 transfer 9-4
 forward alias 9-3
 group alias
 creation 9-4
 handshake message 9-10
 install option 9-4
 LOG file
 connection error location 9-9
 contents 9-9
 machine alias 9-3
 machine name
 choice 9-1
 file entry 9-5
 saving 9-1
 modification 9-11

Micnet network (*continued*)

- netutil program
 - information required 9-1
 - install option 9-4
 - network building 9-4
 - restore option 9-4
 - restore option 9-7
 - save option 9-4
 - save option 9-7
 - start option 9-8
 - stop option 9-10
- planning 9-1
- restore option 9-4
- restore option 9-7
- save option 9-4
- save option 9-7
- serial line
 - assignment 9-2
 - name entry 9-5
 - transmission speed 9-3
- serial topology
 - description 9-2
- standard alias 9-3
- star topology
 - description 9-2
- start option 9-8
- startup
 - procedure 9-8
- stop option 9-10
- stopping 9-10
- testing 9-9
- topology
 - map 9-2
 - types 9-2
- transmission speed
 - assignment 9-3
 - file entry 9-5
- mkfs command
 - block number A-2
 - block size A-1
 - creating file systems 4-1
 - file system creation 4-1
 - gap number A-2
 - special filename A-1
- mkuser program
 - creating a user account 3-1
 - stopping 3-1
- mkuser
 - passwd 3-2
 - shell type 3-2
- /mnt directory, mounted file systems 4-2
- /mnt directory, mounted file systems B-2
- Modes of operation, described 2-2
- more command 8-4
- mount command
 - file system mounting 4-1
 - file system mounting 4-2
 - special filename A-1

mount command (*continued*)

N

- netutil program, install option A-2
- netutil program See Micnet network
- New user 3-1
- newgrp command 3-6
- Normal operation mode 2-2
- Normal operation
 - stopping 2-3

O

- Operating system
 - loading 2-1
- Out of space message 8-3

P

- passwd command 3-3
- Password
 - change procedure 3-3
 - complexity, system access security 4-8
 - forgotten 8-2
 - new user 3-2
 - user account 3-1
- Periodic backup See File system
- Permissions
 - change 4-6
 - description 4-4
 - execute permission 4-5
 - fields 4-5
 - group permissions 4-4
 - initial assignment 4-6
 - levels 4-4
 - no permission 4-5
 - other permissions 4-4
 - read permission 4-5
 - search permission 4-5
 - special files 4-9
 - user permissions 4-4
 - write permission 4-5
- PID
 - killing, lineprinter freeing 8-2
 - killing, runaway process stopping 8-2
- Print queue
 - lockup, freeing 8-1
- Process
 - runaway 8-2
 - stopping 8-2
- .profile file
 - contents 8-5
 - modification 8-4

Index

profile file (*continued*)
 modification 8-5
 removal 3-9
 Program
 runaway process 8-2
 start, initialization files 8-4
 termination, core file placement 5-3

Q

quot command
 block ownership display 5-2
 block size A-1

R

rcp command 9-1
 Read permission 4-5
 remote command 9-1
 rm command 8-3
 rmuser command
 limitations 3-9
 stopping 3-9
 user account removal 3-8
 root
 directory backup 6-1
 directory contents B-1
 super-user login name 2-2
 symbol (/) 4-1

S

Search permission 4-5
 Serial line
 baud rate setting
 procedure 7-4
 Micnet network
 assignment 9-2
 name entry 9-5
 transmission speed 9-3
 Serial lines A-2
 shutdown command 2-3
 Shutdown
 improper shutdown
 file check 5-5
 Slash (/), root symbol 4-1
 Special file
 disabling command 7-2
 enable command 7-2
 protection 4-9
 security 4-9
 Special filename A-1
 Starting the system 2-1
 Stopping the system 2-2
 Super-user password
 secrecy 4-8

Index

Super-user
 account 1-1
 leaving the account 2-2
 log in 2-2
 login name (root) 2-2
 password 1-1
 precautions 2-2
 prompt (#) 2-2
 restricted use 1-1
 special file access 4-9
 sysadmin program
 creating backups 6-2
 description 6-1
 file restoration 6-3
 file restoration 8-3
 listing backups 6-2
 System console
 terminal disabling 7-2
 System maintenance mode
 stopping 2-3
 System manager
 backups 6-1
 duties 1-1
 file access 4-1
 file system maintenance 5-1
 free space maintenance 5-1
 initialization files modification 8-4
 Micnet network maintenance 9-1
 super-user account 1-1
 system maintenance mode 2-2
 user account creation, maintenance 3-1
 System wide message 5-2
 System
 access security 4-8
 accounts 3-9
 adding disk storage 4-1
 administration directory B-3
 cleaning the file system 2-1
 inoperable system restoration 8-3
 maintenance 1-1
 maintenance account 1-1
 maintenance mode 2-2
 physical security 4-8
 problems, fixing 8-1
 reinstallation
 security 4-8
 starting 2-1
 stopping 2-2

T

tar command
 creating copies 6-4
 description 6-1
 restoring copies from disk 6-4
 temporary file
 removal 5-3

TERM variable 7-2

Terminal setting

tset

setting termcap capabilities 7-3

Terminal

connection 7-1

disabling 7-2

enabling 7-2

lockup, runaway process 8-2

nonechoing terminal 8-1

type setting

initialization files 8-4

procedure 7-2

Time

file access 5-3

/tmp directory contents B-2

tset

setting termcap capabilities 7-3

tty line See Serial line

U

umount command 4-1

umount command 4-3

User account

adding 3-1

comments 3-2

directory removal 3-8

file removal 3-8

log in name 3-2

profile file modification 8-5

removal

procedure 3-8

User

block ownership display 5-2

changing the ID 3-7

group See Group

ID 3-7

log in group See Log in group

new user 3-1

password See Password

permissions See Permissions

/usr directory contents B-3

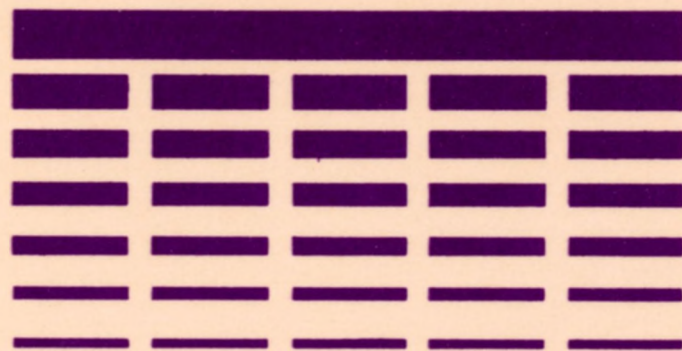
W

wall command 5-2

Write permission 4-5

XENIX™

user's guide



user's guide

TANDY®

The XENIX®
Operating System

User's Guide

Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc. nor Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© 1983, 1984 Microsoft Corporation
© 1984, 1985 The Santa Cruz Operation, Inc.
Licensed to Tandy Corporation

XENIX is a registered trademark of Microsoft Corporation.
MS is a trademark of Microsoft Corporation.

Document Number: G-2-14-85-1.3/1.0

Acknowledgments

This manual builds on the writing of many others. In many cases, the content here is identical, in whole or in part, to papers and manuals written at Bell Laboratories. In particular, Chapters 2 and 5 are adapted from papers written by Brian Kernighan. Chapter 6 is adapted from a document written by Kurt Shoens at the University of California at Berkeley. Chapter 7 is adapted from documents written by G. A. Snyder, J. R. Mashey, and S. R. Bourne. Chapter 8 from a paper written by Robert Morris and Lorinda Cherry. The work of those mentioned above, and countless others, is gratefully acknowledged.

Contents

1 Introduction

- 1.1 Overview 1-1
- 1.2 The XENIX System 1-1
- 1.3 The XENIX Working Environment 1-1
- 1.4 Using This Manual 1-2

2 Demonstration

- 2.1 Introduction 2-1
- 2.2 Before You Log In 2-1
- 2.3 Logging In 2-1
- 2.4 Typing Commands 2-1
- 2.5 Mistakes in Typing 2-3
- 2.6 Read-Ahead and Type-Ahead 2-3
- 2.7 Strange Terminal Behavior 2-3
- 2.8 Stopping a Program 2-3
- 2.9 Logging Out 2-4

3 Basic Concepts

- 3.1 Introduction 3-1
- 3.2 Files 3-1
- 3.3 File Systems 3-3
- 3.4 Naming Conventions 3-3
- 3.5 Commands 3-6
- 3.6 Input and Output 3-8

4 Tasks

- 4.1 Introduction 4-1
- 4.2 Gaining Access to the System 4-1
- 4.3 Configuring Your Terminal 4-2
- 4.4 Editing the Command Line 4-3
- 4.5 Manipulating Files 4-3
- 4.6 Manipulating Directories 4-8
- 4.7 Moving in the File System 4-10
- 4.8 Using File and Directory Permissions 4-11
- 4.9 Processing Information 4-14
- 4.10 Controlling Processes 4-17
- 4.11 Getting Status Information 4-18
- 4.12 Using the Lineprinter 4-19
- 4.13 Communicating with Other Users 4-20
- 4.14 Using the System Clock and Calendar 4-21
- 4.15 Using the Automatic Reminder Service 4-22
- 4.16 Using Another User's Account 4-22
- 4.17 Calculating 4-22

5 Vi: A Text Editor

- 5.1 Introduction 5-1
- 5.2 Demonstration 5-1
- 5.3 Editing Tasks 5-12
- 5.4 Solving Common Problems 5-36
- 5.5 Setting Up Your Environment 5-37
- 5.6 Summary of Commands 5-42

6 Mail

- 6.1 Introduction 6-1
- 6.2 Demonstration 6-2
- 6.3 Basic Concepts 6-3
- 6.4 Using Mail 6-7
- 6.5 Commands 6-10
- 6.6 Leaving Compose Mode Temporarily 6-16
- 6.7 Setting Up Your Environment: The .mailrc File 6-20
- 6.8 Using Advanced Features 6-23
- 6.9 Quick Reference 6-25

7 The Shell

- 7.1 Introduction 7-1
- 7.2 Basic Concepts 7-1
- 7.3 Redirecting Input and Output 7-4
- 7.4 Shell Variables 7-7
- 7.5 The Shell State 7-10
- 7.6 A Command's Environment 7-11
- 7.7 Invoking the Shell 7-12
- 7.8 Passing Arguments to Shell Procedures 7-12
- 7.9 Controlling the Flow of Control 7-13
- 7.10 Special Shell Commands 7-21
- 7.11 Creation and Organization of Shell Procedures 7-22
- 7.12 More About Execution Flags 7-23
- 7.13 Supporting Commands and Features 7-24
- 7.14 Effective and Efficient Shell Programming 7-29
- 7.15 Shell Procedure Examples 7-31
- 7.16 Shell Grammar 7-38

8 BC: A Calculator

- 8.1 Introduction 8-1
- 8.2 Demonstration 8-1
- 8.3 Tasks 8-3
- 8.4 Language Reference 8-10

9 Building a Uucp System

- 9.1 Introduction 9-1
- 9.2 Uucp - System to System File Copy 9-1
- 9.3 Uux - System To System Execution 9-3

- 9.4 Uucico - Copy In, Copy Out 9-5
- 9.5 Uuxqt - Uucp Command Execution 9-7
- 9.6 Uulog - Uucp Log Inquiry 9-8
- 9.7 Uuclean - Uucp Spool Directory Cleanup 9-8
- 9.8 Security 9-8
- 9.9 Installing a Uucp System 9-9
- 9.10 Maintaining the System 9-12

10 The C-Shell

- 10.1 Introduction 10-1
- 10.2 Invoking the C-shell 10-1
- 10.3 Using Shell Variables 10-2
- 10.4 Using the C-Shell History List 10-3
- 10.5 Using Aliases 10-5
- 10.6 Redirecting Input and Output 10-6
- 10.7 Creating Background and Foreground Jobs 10-6
- 10.8 Using Built-In Commands 10-7
- 10.9 Creating Command Scripts 10-8
- 10.10 Using the argv Variable 10-8
- 10.11 Substituting Shell Variables 10-9
- 10.12 Using Expressions 10-10
- 10.13 Using the C-Shell: A Sample Script 10-11
- 10.14 Using Other Control Structures 10-13
- 10.15 Supplying Input to Commands 10-13
- 10.16 Catching Interrupts 10-14
- 10.17 Using Other Features 10-14
- 10.18 Starting a Loop at a Terminal 10-14
- 10.19 Using Braces with Arguments 10-15
- 10.20 Substituting with Arguments 10-16
- 10.21 Special Characters 10-16

11 Using The Visual Shell

- 11.1 What is the Visual Shell? 11-1
- 11.2 Getting Started with the Visual Shell 11-1
- 11.3 The Visual Shell Screen 11-2
- 11.4 Visual Shell Reference 11-4

Appendix A Ed

- A.1 Introduction A-1
- A.2 Demonstration A-1
- A.3 Basic Concepts A-1
- A.4 Tasks A-2
- A.5 Context and Regular Expressions A-23
- A.6 Speeding Up Editing A-35
- A.7 Cutting and Pasting with the Editor A-38
- A.8 Editing Scripts A-40
- A.9 Summary of Commands A-41

Chapter 1

Introduction

- 1.1 Overview 1
- 1.2 The XENIX System 1
- 1.3 The XENIX Working Environment 1
- 1.4 Using This Manual 2

1.1 Overview

This manual introduces the XENIX system by explaining the fundamental concepts and software needed to use it effectively. The XENIX system is an improved and enhanced version of the UNIX SYSTEM III_(TM) from Bell Laboratories. It is intended for use in schools, corporations, laboratories and small office environments. The system is well known as a productive environment for software development and has been used for many years as a text processing environment.

1.2 The XENIX System

The XENIX system consists of a general-purpose multiuser operating system and over one hundred utilities and application programs. In addition to the XENIX Operating System described in this manual, two other XENIX system packages are available: the XENIX Development System and the XENIX Text Processing System.

1.3 The XENIX Working Environment

The XENIX system is built around the XENIX operating system. The purpose of an operating system is to efficiently organize and control the resources of a computer so that they can be used by real people. These resources include memory, disks, lineprinters, terminals, and any other peripheral devices connected to the system. The heart of the XENIX system is a “multiuser” and “multitasking” operating system. A multiuser system permits several users to use a computer simultaneously, thus providing lower cost in computing power per user. A multitasking system permits several programs to run at the same time and increases productivity because multiple programs can run simultaneously rather than in sequence.

Because UNIX (and thus XENIX) has been accepted as a standard for “high-end” operating systems, a great deal of software is available for this environment. In addition, XENIX is a bridge to the MS-DOS operating system, the most widely used 16-bit operating system in the world. For systems that support MS-DOS, XENIX provides commands that let you access MS-DOS format files and disks. The XENIX system also includes several widely praised enhancements developed at the University of California at Berkeley, and a visual interface similar to other Microsoft productivity tool interfaces.

Other characteristics of the XENIX system include:

- A powerful command language for programming XENIX commands. Unlike other interactive command languages, the XENIX “shell” is a full programming language.
- Simple and consistent naming conventions. Names can be used absolutely, or relative to any directory in the file system.
- Device-independent input and output: each physical device, from interactive terminals to main memory, is treated like a file, allowing uniform file and device input and output.
- A set of related text editors, including a full screen editor.
- Flexible text processing facilities. In XENIX, commands exist to find and extract patterns of text from files, to compare and find differences between files, and to search through and compare directories. Text formatting, typesetting, and spelling error-detection facilities, as well as a facility for formatting and typesetting complex tables and equations are also available.

- A sophisticated “desk-calculator” program.
- Mountable and dismountable file systems that permit addition of floppy disks to the file system.
- A complete set of flexible directory and file protections that allows all combinations of read, write, and execute access for the owner of each file or directory, as well as for groups of users.
- Facilities for creating, accessing, moving, and processing files and directories in a simple and uniform way.

1.4 Using This Manual

This manual is organized as follows:

Chapter 1: Introduction

This chapter gives an introduction and overview of the XENIX system.

Chapter 2: Demonstration

This chapter gives you hands-on experience in using the XENIX system.

Chapter 3: Basic Concepts

This chapter explains the fundamental concepts that you need to understand before you begin to use the system. Included here are sections on the file system, naming conventions, commands, and input and output.

Chapter 4: Tasks

This chapter explains how to perform everyday tasks using appropriate XENIX commands.

Chapter 5: Vi

This chapter explains how to use the screen editor, **vi**.

Chapter 6: The Shell

This chapter describes use of the shell command interpreter and how to write procedures that can be executed by the shell interpreter.

Chapter 7: Mail

This chapter describes the XENIX mail facility and explains how to send and receive mail.

Chapter 8: BC: A Calculator

This chapter explains how to use **BC**, a sophisticated calculator program.

Chapter 9: Building a UUCP System

This chapter explains how to set up a system to permit communication between XENIX and/or UNIX systems using dial-up communication lines.

Chapter 10: The C-Shell

This chapter describes how to use **csh**. It covers the syntax and function of C-shell commands and features, and how to create shell procedures.

Chapter 11: Using The Visual Shell

This chapter describes the use and behavior of the Visual Shell, which is a menu-driven XENIX shell. This chapter assumes the reader is familiar with some general XENIX concepts, but **vsh** can be used by first-time users.

Appendix A: Ed

This chapter explains how to use the editor, **ed**.

This manual does not attempt to give information about installing, managing, and maintaining the system, nor does it discuss document preparation, software development, or many of the specialized utilities available in other XENIX system products. These subjects are covered in the following manuals:

The XENIX Installation Guide

This guide describes how to install and set up the XENIX system on your computer.

The XENIX Operations Guide

This manual is a guide to managing and maintaining the entire system.

The XENIX Reference Manual

This manual is a comprehensive command reference. A concise but complete description of each command is available here. It includes manual pages for Commands(C) and Miscellaneous(M).

The XENIX Programmer's Guide

This manual discusses how to use the programming tools available in the XENIX programming environment. This manual is part of the optional XENIX Development System. It includes the manual page reference section for Subroutines(CP).

The XENIX Programmer's Reference

This manual discusses writing programs that interface to the XENIX operating system. It provides reference to system calls, subroutines, and file formats. This manual is part of the optional XENIX Development System. It includes the manual page reference sections for Sytem Calls(S) and File Formats(F).

The XENIX Text Processing Manual

This manual explains how to use the text processing and text formatting tools and includes the manual pages for Text Commands(CT). It is a part of the XENIX Text Processing System.

Chapter 2

Demonstration

2.1 Introduction	1
2.2 Before You Log In	1
2.3 Logging In	1
2.4 Typing Commands	1
2.5 Mistakes in Typing	3
2.6 Read-Ahead and Type-Ahead	3
2.7 Strange Terminal Behavior	3
2.8 Stopping a Program	3
2.9 Logging Out	4

2.1 Introduction

This chapter contains a demonstration run designed to help you get used to the XENIX system, so that you can quickly start to make effective use of it. It shows you how to log in, how to type at your keyboard, what to do about mistakes in typing, how to enter commands and how to log out.

2.2 Before You Log In

Before you can log in to the system, your name must be added to the XENIX user list. At that time you will be given a login name and a password. You may have to add your name yourself, or someone else may be assigned this task; it all depends on the environment in which your system is used. In any case, see the *XENIX Operations Guide* and *mkuser* (C) for detailed information on adding users.

When you are given an account on the XENIX system you will also receive a user name, a password, and a login directory. Once you have these, all you need is a terminal from which you can log in to the system. XENIX supports most terminals and you should have no problem getting your terminal to work with XENIX. Once again, see the *XENIX Operations Guide* for more information on how to configure your terminal.

2.3 Logging In

Normally the system is sitting idle with a "login:" prompt on the terminal screen. If the system displays nonsense characters when you type, then your terminal is probably receiving information at the wrong speed and you should check your terminal switches. If the switches are set correctly, push the BREAK or INTERRUPT key a few times.

When you get a "login:" message, type your login name, then press RETURN; the system will not do anything until you do. If a password is required, you will be asked for it. The password that you type does not appear on the screen. This prevents others from viewing it. Don't forget to press RETURN after you type your password.

A successful login produces a "prompt character", a single character that indicates the system is ready to accept commands. The prompt is usually a dollar sign (\$) or a percent sign (%). You may also get a login message such as:

you have mail

telling you that another system user has sent you mail.

2.4 Typing Commands

Once the prompt character appears, the system is ready to respond to commands typed at the terminal. Try typing

date

followed by RETURN. The system responds by displaying something like:

Mon Jun 16 14:17:10 EST 1983

Don't forget to press the RETURN key after the command, or nothing will happen. The RETURN key won't be mentioned again, but don't forget -- it has to be entered at the end of each command line. On some terminals RETURN may be labeled "ENTER" or "CR", but in all cases, the key performs the same function.

Another command you might try is **who**, which lists the names of everyone who is logged in to XENIX. A typical display from the **who** command might look something like this:

XENIX User's Guide

```
you  console Jan 16 14:00
joe  tty01  Jan 16 09:11
ann  tty02  Jan 16 09:33
```

The time, given in the fourth column, indicates when the user logged in; *ttynn* is the system name for each user's terminal, where *nn* is a unique two-digit number. The console is the special name of the master terminal that is the default for most operations.

If you make a mistake typing the command name, you will see a message on your screen. For example, if you type:

```
whom
```

the system responds with the message:

```
whom: not found
```

Note that case is significant in XENIX. The commands

```
who
```

```
and
```

```
WHO
```

are not the same; this differs from some operating systems, where case doesn't matter.

Now try displaying a message on your screen using the **echo** command. Type

```
echo hello world
```

The **Echo** command does what its name implies and echoes the rest of the command line to your terminal:

```
hello world
```

Now try this:

```
echo hello world >greeting.file
```

This time the **echo** command sends its output to a new file named *greeting.file*, instead of to your terminal. Note the use of the greater-than sign (>) to "redirect" the output of the command. Now type

```
lc
```

to list just the name of the file, *greeting.file*. To look at the contents of *greeting.file*, display it by typing:

```
cat greeting.file
```

Here "cat" stands for concatenate. One purpose of the **cat** command is to combine the contents of several files (that is, "concatenate") and put them in some new file. However, since your terminal display is treated like any other file in XENIX, **cat** is most commonly used to display the contents of files on the screen. Therefore the above command sends the following output to your terminal screen:

```
hello world
```

To remove *greeting.file*, type:

```
rm greeting.file
```

Note that XENIX command names are often shortened to mnemonic names. For example, **cp** is short for "copy", **ls** is short for "list", **rm** is short for "remove", **cat** is short for "concatenate", **mkdir** is short for "make directory", and **chmod** is short for "change mode".

2.5 Mistakes in Typing

If you make a mistake in typing while entering a command, there are two ways to edit the line, provided you have not yet pressed RETURN. Pressing the BKSP key causes the last character typed to be erased. Backspacing with the BKSP key can erase characters back to the beginning of the line, but not beyond. Thus, if you type badly, you can correct as you go. For example, typing

```
ddBKSPateRETURN
```

is the same as

```
dateRETURN
```

The XENIX kill character, CNTRL-U, erases all of the characters typed so far on the current input line. So, if the line is irretrievably fouled up, type CNTRL-U and start the line over.

If you must enter a BKSP or CNTRL-U as part of the text, precede it with a backslash (\), so that the character loses its special ““erase”” meaning. To enter a BKSP or CNTRL-U in text, type “\BKSP” or “\CNTRL-U”. The system always prints a new line on your terminal after your CNTRL-U, even if preceded by a backslash. Nevertheless, the CNTRL-U will have been recorded.

To erase a backslash, backspace twice with the BKSP key, as in “\BKSPBKSP”. The backslash is used extensively in XENIX to indicate that the following character is in some way special. Note that the functions performed by BKSP and CNTRL-U are available on all XENIX systems; however, the keys used to perform these functions may vary and can be set by the user with *stty*(C).

2.6 Read-Ahead and Type-Ahead

XENIX has full read-ahead, which means that you can type as fast as you want, whenever you want, and XENIX will remember what you have typed. If you enter any text while a command is displaying text on the screen, your input characters appear intermixed with the output characters on the screen, but they are stored away and interpreted in the correct order. Therefore, you can type several commands (i.e., “type ahead”) one after another without waiting for the first to finish. Note that this doesn’t work when you log in; type-ahead does not work until *after* you have entered your password and the dollar sign (\$) prompt appears.

2.7 Strange Terminal Behavior

Occasionally, your terminal may act strangely. You can often fix such behavior by either turning your terminal off, then quickly turning it back on, or logging out and logging back in; this will reset your terminal characteristics. If logging out and back in doesn’t work, read the description of the command *stty*(C) in the *XENIX Reference Manual* for more information about setting terminal characteristics.

2.8 Stopping a Program

You can abort the execution of most programs and commands by pressing the INTERRUPT key (perhaps called DEL, DELETE, CNTRL-C, or RUBOUT on your terminal). The BREAK key found on many terminals can also be used. Inside some programs, like most text editors, typing INTERRUPT stops whatever the program is doing without aborting the program itself. Throughout this manual, when we say “send an interrupt” we mean press the INTERRUPT key.

2.9 Logging Out

To end a session with XENIX, you must log out. This is done by typing CNTRL-D as the first character on a line. It is not sufficient just to turn off the terminal, since this does not log you out. Some programs can also be ended by typing CNTRL-D, so beware.

Chapter 3

Basic Concepts

3.1 Introduction	1
3.2 Files	1
3.2.1 Ordinary Files	1
3.2.2 Special Files	1
3.2.3 Directory files	1
3.2.4 Directory Structure	2
3.3 File Systems	3
3.4 Naming Conventions	3
3.4.1 Filenames	3
3.4.2 Pathnames	4
3.4.3 Sample Names	4
3.4.4 Special Characters	5
3.5 Commands	6
3.5.1 Command Line	6
3.5.2 Syntax	7
3.6 Input and Output	8
3.6.1 Redirection	8
3.6.2 Pipes	9

3.1 Introduction

This chapter will give you an understanding of the basic concepts you need to function in the XENIX environment. After reading this chapter you should understand how the system's files, directories, and devices are organized and named, how commands are entered, and how a command's input and output can be manipulated. This chapter begins with a discussion of files.

3.2 Files

The file is the fundamental unit of the XENIX file system. In XENIX there are really three different types of files: ordinary files (what we usually mean when we say "file"), directories, and special files. Each of these types of files is described below.

3.2.1 Ordinary Files

Ordinary files typically contain textual information such as documents, data, or program sources. Executable binary files are also of this type. An ordinary file is simply a named concatenation of 8-bit bytes. Whether these bytes are interpreted as text characters, binary instructions, or program statements is up to the programs that examine them. Every ordinary file has the following attributes:

- A filename (not necessarily unique)
- A unique system number called an inode number
- A size in bytes
- A time of creation
- A time of modification
- A time of last access
- A set of access permissions

Files can be protected by assigning appropriate access permissions to assure privacy and security. This is done by providing read-write-execute permissions to files so that the user can control access by the owner, by a group of users, and by anyone else. By default, the owner of a file is its creator. The owner can read the file or write to it. By default, other users can read a file owned by another, but not write to it. File permissions can be altered with the **chmod** command. This command is discussed in Chapter 4 of this manual.

3.2.2 Special Files

Special files correspond to physical devices such as hard and floppy disks, lineprinters, terminals, and system memory. They are called "device special files". These files are not discussed in this manual.

3.2.3 Directory files

Directory files are read-only files containing information about the files or directories that are conceptually (but not physically) contained within them. This information consists of the name and inode number of each file or directory residing within the given directory. An inode number

is a unique number associated with any given file. *All* files on the system have inode numbers. A name/inode number pair is called a link. The **ls** command is used to examine directory files and to list the information about the files conceptually within the named directory. With the inode number, the **ls** command can also find other information about a file.

The nesting of directories inside other directories is the way in which XENIX implements its characteristic tree-structured directory system. Directories are discussed further in the next section.

Like ordinary files, directories can be protected by assigning appropriate access permissions to assure privacy and security. This is done by giving read-write-search permissions to directories so that the user can control directory access by the owner, by a group of users, and by anyone else. Write permission determines whether files can be added or removed from a directory. By default, the owner of a directory is its creator, and the owner can read, create or remove files within that directory. Similarly by default, a user can read files within the directory of another, but not add or remove files. As with file permissions, directory permissions can be altered with the **chmod** command. Default permissions can be altered with the **umask** command.

3.2.4 Directory Structure

With multiple users and multiple projects, the number of files in a file system can proliferate rapidly. Fortunately, as mentioned earlier, XENIX organizes all files into a tree-structured directory hierarchy. This tree structure should be thought of as a physical world in which the user can move from place to place. "Places" are directories. Each user of the system has his own personal directory. Within that directory, the user may have directories or other subdirectories owned and controlled only by the user.

When you log in to XENIX, you are "in" your directory. Unless you take special action when you create a file, the new file is created in your working directory. This file is unrelated to any other file of the same name in someone else's directory.

A diagram of part of a typical user directory is shown in Figure 3-1.

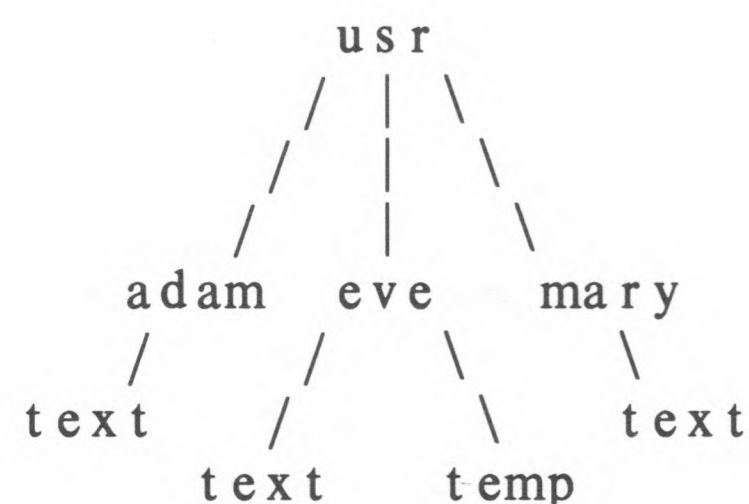


Figure 3-1. A Typical User Directory

In Figure 3-1, the *usr* directory contains each user's own personal directory. Notice that Mary's file named *text* is unrelated to Eve's. This is not important if all the files of interest are in Eve's directory, but if Eve and Mary work together, or if they work on separate but related projects, this division of files becomes handy indeed. For example, Mary could print Eve's text by typing:

```
pr /usr/eve/text
```

Similarly, Eve could find out what files Mary has by typing:

```
lc /usr/mary
```


3.3 File Systems

A file system is a set of files organized in a certain way. In XENIX, this set of files consists of all available resources including data files, directories, programs, lineprinters, and disks. Thus, the XENIX file system is a system for accessing all system resources.

To logically structure the resources of the system, the XENIX file system is organized hierarchically in an inverted “tree structure”. See Figure 3-2 for an illustration of a typical tree-structured file system. In this typical tree of files, the root of the tree is at the top and branches of the tree grow downward. Directories correspond to nodes in the tree; ordinary files correspond to “leaves”. If a directory contains a downward branch to other files or directories, then those files and directories are “contained” in the given directory. It is possible to name any file in the system by starting at the root (where the root is at the top) and traveling down any of the branches to the desired file. Similarly, you can specify any file in the system, relative to any directory. Specification of these files depends on a knowledge of the XENIX naming conventions, discussed in the next section.

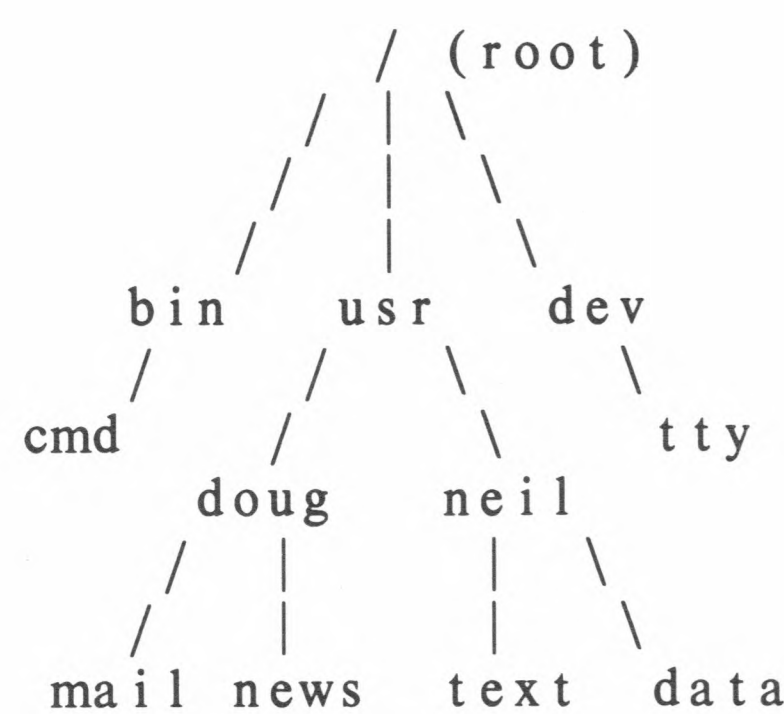


Figure 3.2. A Typical File System

In the typical tree-structured file system of Figure 3-1, the “tree” grows downward. The names *bin*, *usr*, *dev*, *doug*, and *neil* all represent directories, and are all nodes in the tree. In XENIX the name of the root directory is given the one-character name, “/”. The names *mail*, *news*, *text*, and *data* all represent normal data files, and are all “leaves” of the tree. Note that the file *cmd* is the name of a command that can be executed. The name *tty* represents a terminal and is also represented in the tree.

3.4 Naming Conventions

Every single file, directory, and device in XENIX has both a filename and an absolute pathname. This pathname is a map of the file or directory’s location in the system. The absolute pathname is unique to all names in the system; filenames are unique only within directories and need not be unique system-wide. This is similar to someone whose “global” name is John Albert Smith in a telephone directory, but who may be listed simply as John in an office phone list.

3.4.1 Filenames

A simple filename is a sequence of one to fourteen characters other than a slash (/). Every single file, directory, and device in the system has a filename. Filenames are used to uniquely identify directory contents. Thus, no two filenames in a directory may be the same. However, filenames in different directories may be identical.

Although you can use almost any character in a filename, it is best to confine filenames to the alphanumeric characters and the period. Other characters, especially control characters, are discouraged for use in filenames. When a filename contains an initial period, it is “hidden”, and is not displayed by the *lc* command. However the *ls* command will display the hidden files. The

dash (—) is used in specifying command options, and should be avoided when naming files. In addition, the question mark (?), the asterisk (*), brackets ([and]), and all quotation marks should *never* be used in filenames, since they are treated specially when entering commands.

3.4.2 Pathnames

A pathname is a sequence of directory names followed by a simple filename, each separated from the previous name by a slash. If a pathname begins with a slash, it specifies a file that can be found by beginning a search at the *root* of the entire tree. Otherwise, files are found by beginning the search at the user's *current directory* (also known as the *working directory*). The current directory should be thought of as your location in the file system. Think of it as a physical place. When you change your current directory you are moving to some other directory or place in the file system.

A pathname beginning with a slash is called a *full* (or *absolute*) *pathname* because it does not vary with regard to the user's current directory. A pathname *not* beginning with a slash is called a *relative pathname*, because it specifies a path relative to the current directory. The user may change the current directory at any time by using the **cd** command.

3.4.3 Sample Names

Some sample names follow:

/	The absolute pathname of the root directory of the entire file system.
/bin	The directory containing most of the frequently used XENIX commands.
/usr	The directory containing each user's personal directory. The subdirectory, <i>/usr/bin</i> contains frequently used XENIX commands not in <i>/bin</i> .
/dev	The directory containing files corresponding to physical devices (e.g., terminals, lineprinters, and disks).
/dev/console	The name of the system master terminal.
/dev/tty	The name of the user's terminal.
/lib	The directory containing files used by some standard commands.
/tmp	This directory contains temporary scratch files.
/usr/joe/project/A	A typical full pathname; this one happens to be a file named <i>A</i> in the directory named <i>project</i> belonging to the user named <i>joe</i> .
bin/x	A relative pathname; it names the file <i>x</i> in subdirectory <i>bin</i> of the current working directory. If the current directory is <i>/</i> , it names <i>/bin/x</i> . If the current directory is <i>/usr/joe</i> , it names <i>/usr/joe/bin/x</i> .
file1	Name of an ordinary file in the current directory.

When using the XENIX system, each user resides "in" a directory called the current directory. All files and directories have a "parent" directory. This directory is the one immediately above, which "contains" the given file or directory. The XENIX file system provides special shorthand notations for this directory and for the current directory:

- . The shorthand name of the current directory. Thus *./filexxx* names the same file as *filexxx*, if such a file exists in the current directory.
- .. The shorthand name of the current directory's parent directory. The shorthand name *../..* refers to the directory that is two levels "above" the current directory

3.4.4 Special Characters

XENIX provides a pattern-matching facility for specifying sets of filenames that match particular patterns. For example, examine the problem that occurs when naming the parts of a large document, such as a book. Logically, it can be divided into many small pieces such as chapters or sections. Physically, it must be divided too, since the XENIX editor *vi* cannot handle really big files. Thus, you should divide a large document into several files. The points at which the document is divided should follow a logical order. You might have a separate file for each chapter:

```
chap1
chap2
...
```

Or, if each chapter is broken into several files, you might have:

```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

You can then tell at a glance where a particular file fits into the whole.

There are other advantages to a systematic naming convention that are not so obvious. What if you want to print the whole book on the lineprinter? You could type

```
lpr chap1.1 chap1.2 chap1.3 ...
```

but you will tire of this quickly and will probably even make mistakes. Fortunately, there is a shortcut: a sequence of names containing a common pattern can be specified with the use of special characters. The special characters discussed in this chapter are:

* Matches zero or more characters

[] Matches any character inside the brackets

? Matches any single character

For example, you can type:

```
lpr chap*
```

The asterisk (*), sometimes called "star" in XENIX, means "zero or more characters of any type", so this translates into "send all files whose names begin with the word "chap" to the lineprinter".

This shorthand notation is not a unique property of the *lpr* command; it can be used in any command. Using this fact, you can list the names of the files in the book by typing:

```
ls chap*
```

This produces


```
chap1.1
chap1.2
chap1.3
...
```

The star is not limited to the last position in a filename; it can be used anywhere and can occur several times. A star by itself matches all filenames not containing slashes or beginning with periods, so

```
cat *
```

displays all files in the current directory on your terminal screen.

The star is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4, and 9. You can say

```
lpr chap[12349]*
```

The brackets ([and]) mean “match any of the characters inside the brackets.” A range of consecutive letters or digits can be abbreviated, so you can also do this with

```
lpr chap[1-49]*
```

(Note that this does *not* match forty-nine filenames, but only five.) Letters can also be used within brackets: “[a-z]” matches any character in the range “a” through “z”.

The question mark (?) matches any single character, so

```
ls ?
```

lists all files that have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter (i.e., *chap1.1*, *chap2.1*, ...).

If you need to turn off the special meaning of any of the special characters (*, ?, and [...]) enclose the entire argument in single quotation marks. For example, the following command will print out only files named “?” rather than all one-character filenames:

```
ls '?'
```

Pattern-matching features are discussed further in Chapter 7, “The Shell.”

3.5 Commands

Commands are used to invoke executable programs. When you type the name of a command, XENIX reads the command line that you have typed, looks for a program with the given name, and then executes the program if it finds it. Command lines may also contain arguments that specify options or files that the program may need. The command line and command syntax are discussed in the next two sections.

3.5.1 Command Line

Whether you are typing commands at a terminal, or XENIX is reading commands from a file, XENIX always reads commands from command lines. The command line is a line of characters that is read by the shell command interpreter to determine what actions to perform. This interpreter, or “shell” as it is known, reads the names of commands from the command line, finds the executable program corresponding to the name of the command, then executes that program. When the program finishes executing, the shell resumes reading the command line. Thus, when you are typing at a terminal, you are editing a line of text called the *command-line buffer* that becomes a command line only when you press RETURN. This command-line buffer can be edited with the BKSP and CNTRL-U keys. Pressing RETURN causes the command-line buffer to be

submitted to the shell as a command line. The shell reads the command line and executes the appropriate command. If you press INTERRUPT before you press RETURN, the command-line buffer is erased. Multiple commands can be entered on a single command line provided they are separated by a semicolon (;). For example, the following command line prints out the current date and the name of the current working directory:

```
date ; pwd
```

Commands can be submitted for processing in “the background” by appending an ampersand (&) to the command line. This mode of execution is similar to “batch” processing on other systems. The main advantage to placing commands in the background is that you can execute other commands from your terminal in the “foreground” while the background commands execute. Thus:

```
du /usr > diskuse&
```

determines the disk usage in the directory */usr*, a fairly time-consuming operation, without tying up your terminal. Note that the output is placed in the file *diskuse* by redirecting output with the greater-than symbol. Redirection is discussed in Section 3.6.1.

3.5.2 Syntax

The general syntax for commands is as follows:

```
cmd [ switches ] [ arguments ] [ filenames ]
```

By convention, command names are lowercase. Switches, also called options, are flags that select various options available when executing the command. They are optional and usually *precede* other arguments and filenames. Switches consist of a dash prefix (–) and an identifying letter. For example, the **ls** command’s **–l** switch (pronounced “minus ell”) specifies a long directory listing and the command

```
ls –r
```

specifies a directory listing in reverse alphabetical order. In some cases, switches can be grouped to form a single switch argument. For example, the command

```
ls –rl
```

is really a combination of two switches, where the **–rl** switch selects the option that lists all files in the directory in both reverse alphabetical order and with the long format.

Sometimes multiple switches must be given separately, as in:

```
copy –v –a source destination
```

Here the **–v** switch specifies the “verbose” option, which reports copying as it happens. The **–a** switch tells the **copy** command to ask the user for confirmation before copying the *source* to the *destination*.

Other arguments, such as search strings, can also be given, as in:

```
grep 'string of text' outfile
```

In the above example,

```
'string of text'
```

is a single argument and is the search string the **grep** command searches for in the file *outfile*. *Filename* is the argument that specifies the name of a file required by the command.

Most commands are executable programs compiled by the C compiler or by some other language compiler. Some commands are executable command files called “shell procedures”. Shell procedures are discussed in Chapter 7, “The Shell.”

3.6 Input and Output

By default, XENIX assumes that terminal input comes from the terminal keyboard and output goes to the terminal screen. To illustrate typical command input and output, type:

```
cat
```

This command now expects input from your keyboard. As input, it accepts as many lines of text as you type until you press CNTRL-D as an end-of-file or end-of-transmission indicator.

For example, type:

```
this is two linesRETURN
of inputRETURN
CNTRL-D
```

When you press CNTRL-D, input ends and output begins. The **cat** command immediately outputs the two lines you typed—since output is sent to the terminal screen by default, that is where the two lines are sent. Thus, the complete session will look like this on your terminal screen:

```
$ cat
this is two lines
of input
this is two lines
of input
$
```

The flow of command input and output can be “redirected” so that input comes from a file instead of from the terminal keyboard and output goes to a file or lineprinter, instead of to the terminal screen. In addition, “pipes” can be created that allow the output from one command to become the input to another. Redirection and pipes are discussed in the next two subsections.

3.6.1 Redirection

In XENIX a file can replace the terminal for either input or output. For example

```
ls
```

displays a list of files on your terminal screen. But if you say

```
ls > filelist
```

a list of your files is placed in the file *filelist* (which is created if it does not exist). The symbol for output redirection, the greater-than sign (>), means “put the output from the command into the following file, rather than display it on the terminal screen.” As another example of output redirection, you can combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 > temp
```

The output append symbol (>>) operates very much like the output redirection symbol, except that it means “add to the end of”. So

```
cat file1 file2 file3 >> temp
```

means “concatenate *file1*, *file2*, and *file3* to the end of whatever is already in *temp*, instead of overwriting and destroying the existing contents.” As with normal output redirection, if *temp* doesn't exist, it is created for you.

In a similar way, the input redirection symbol (<) means “take the input for a program from the following file, instead of from the terminal”. Thus, you could make a script of editing commands and put them into a file called *script*. Then you could execute the commands in the script on a file by typing:


```
ed file <script
```

As another example, you could use **ed** to prepare a letter in the file *letter.txt*, then send it to several people with:

```
mail adam eve mary joe <letter.txt
```

3.6.2 Pipes

One of the major innovations of the XENIX system is the concept of a “pipe”. A pipe is simply a way to connect the output of one command to the input of another, so that the two run as a sequence of commands called a pipeline.

For example:

```
sort frank.txt george.txt hank.txt
```

combines the three files named *frank.txt*, *george.txt*, and *hank.txt*, then sorts the output. Suppose that you want to then find all unique words in these files and view the result. You could type:

```
sort frank.txt george.txt hank.txt >temp1
uniq <temp1 >temp2
more temp2
rm temp1 temp2
```

But this is more work than is necessary. What you want is to take the output of **sort** and connect it to the input of **uniq**, then take the output of **uniq** and connect it to **more**. You would use the following pipe:

```
sort frank.txt george.txt hank.txt | uniq | more
```

The vertical bar character (|) is used between the **sort** and **uniq** commands to indicate that the output from **sort**, which would normally have been sent to the terminal, is to be redirected from the terminal to the standard input of the **uniq** command, which in turn sends its output to the **more** command for viewing.

There are many other examples of pipes. For example

```
ls | pr -3
```

formats and paginates a list of your files in three columns. The program **wc** counts the number of lines, words, and characters in its input, and **who** prints a list of users currently logged on, one per line. Thus,

```
who | wc
```

tells how many people are logged in, and

```
ls | wc
```

counts the number of files in the current directory.

Any program that reads from the terminal keyboard can read from a pipe instead. Any program that displays output to the terminal screen can send input to a pipe. You can have as many elements in a pipeline as you wish.

Chapter 4

Tasks

4.1	Introduction	1
4.2	Gaining Access to the System	1
4.2.1	Logging In	1
4.2.2	Logging Out	1
4.2.3	Changing Your Password	2
4.3	Configuring Your Terminal	2
4.3.1	Changing Terminals	2
4.3.2	Setting Terminal Options	3
4.4	Editing the Command Line	3
4.4.1	Entering a Command Line	3
4.4.2	Erasing a Command Line	3
4.4.3	Halting Screen Output	3
4.5	Manipulating Files	3
4.5.1	Creating a File	4
4.5.2	Displaying File Contents	4
4.5.3	Combining Files	5
4.5.4	Moving a File	5
4.5.5	Renaming a File	6
4.5.6	Copying a File	6
4.5.7	Deleting a File	6
4.5.8	Finding Files	7
4.5.9	Linking a File to Another File	7
4.6	Manipulating Directories	8
4.6.1	Printing the Name of Your Working Directory	8
4.6.2	Listing Directory Contents	8
4.6.3	Creating a Directory	9
4.6.4	Removing a Directory	9
4.6.5	Renaming a Directory	10
4.6.6	Moving a Directory	10
4.6.7	Copying a Directory	10
4.7	Moving in the File System	10
4.7.1	Finding Out Where You Are	10
4.7.2	Changing Your Working Directory	11
4.8	Using File and Directory Permissions	11
4.8.1	Changing Permissions	13
4.8.2	Changing Directory Search Permissions	14
4.9	Processing Information	14
4.9.1	Comparing Files	14
4.9.2	Echoing Arguments	14
4.9.3	Sorting a File	15

4.9.4	Searching for a Pattern in a File	15
4.9.5	Counting Words, Lines, and Characters	16
4.9.6	Delaying a Process	16
4.10	Controlling Processes	17
4.10.1	Placing a Process in the Background	17
4.10.2	Killing a Process	18
4.11	Getting Status Information	18
4.11.1	Finding Out Who is on the System	18
4.11.2	Finding Out What Processes Are Running	19
4.11.3	Getting Lineprinter Information	19
4.12	Using the Lineprinter	19
4.12.1	Sending a File to the Lineprinter	19
4.12.2	Getting Lineprinter Queue Information	20
4.13	Communicating with Other Users	20
4.13.1	Sending Mail	20
4.13.2	Receiving Mail	20
4.13.3	Writing to a Terminal	21
4.14	Using the System Clock and Calendar	21
4.14.1	Finding Out the Date and Time	21
4.14.2	Displaying a Calendar	21
4.15	Using the Automatic Reminder Service	22
4.16	Using Another User's Account	22
4.17	Calculating	22

4.1 Introduction

This chapter explains how to perform common tasks on XENIX. The individual commands used to perform these tasks are discussed more thoroughly in the *XENIX Reference Manual*.

4.2 Gaining Access to the System

To use the XENIX system, you must first gain access to it by logging in. When you log in you are placed in your own personal working area. Logging in, changing your password, and logging out are described below.

4.2.1 Logging In

Before you can log in to the system, you must be given a system “account”. Your name must be added to the user list, and you must be given a password and a mailbox.

Depending on how your system is administered, you may have to add your name to the user list yourself, or someone else may be assigned this task. If you must add your own account to the system, see the *XENIX Operations Guide* and *mkuser(C)* in the *XENIX Reference Manual* for more information. This section assumes your account has already been set up.

Normally, the system sits idle and the prompt “login:” appears on the terminal screen. If your screen is blank, or displays nonsense characters, press the INTERRUPT key a few times.

When the “login:” prompt appears, follow these steps:

1. Type your login name and press RETURN. If you make a mistake, press CNTRL-U to start the line again. After you press RETURN the word “Password:” appears on your screen.
2. Type your password carefully, then press RETURN. The letters do not appear on your screen as you type, and the cursor does not move. If you make a mistake, press RETURN to restart the login procedure.

If you have typed your login name and password correctly the “prompt character” appears on the screen. This is usually a dollar sign(\$). The prompt tells you that the XENIX system is ready to accept commands from the keyboard.

If you make a mistake, the system displays the message:

```
Login incorrect  
login:
```

If you get this message, follow the above procedure again. You must type all the letters of your user name and password correctly before you are given access to the system; XENIX does not allow you to correct your mistakes when typing your password.

Depending on how your system is set up, after you log in you may see a “banner” that says something like “Welcome to XENIX”, or an announcement that is of interest to all users.

4.2.2 Logging Out

The logout procedure is simple—all you need to do is press

```
CNTRL-D
```

alone on a line. In general, CNTRL-D signifies the end-of-file in XENIX, and is often used within programs to signal the end of input from the keyboard. In such cases, CNTRL-D will *not* log you out; it will simply terminate input to a particular program if you are within that program. This

means that it may sometimes be necessary to press CNTRL-D several times before you can log yourself out. For example, if you are in the **mail** program you must press CNTRL-D once to exit the mail program, then again to log out.

4.2.3 Changing Your Password

To prevent unauthorized users from gaining access to the system, each authorized user must have a password. When you are first given an account on a XENIX system you are assigned a password by the system administrator. Some XENIX systems require you to change your password at regular intervals. Whether yours does or not, it is a good idea to change your password regularly to maintain system security. This section tells you how to change your password.

Use the **passwd** command to change your password. Follow these steps:

1. Type

`passwd`

and press RETURN. The following message appears:

Changing password for *user*:

Old password:

2. Carefully type your old password. It is not displayed on the screen. If you make a mistake, press RETURN. The message "Sorry" appears, then the system prompt. Begin again with step 1.

3. When you have typed your old password the message

New password:

appears. Type in your new password and press RETURN.

4. The message

Retype new password:

appears. Type your new password again. If you make a mistake, press RETURN. The message

Mismatch -- password unchanged

appears, and you must begin again with step 1. When you have completed the procedure, the system prompt appears.

4.3 Configuring Your Terminal

On most systems, the standard console terminal is already configured for use with XENIX. However, other terminals of various types may be connected to a XENIX system. In these cases it is important to know how to set terminal options and how to specify the terminal you are using. You may also want to change the standard configuration of the standard console terminal. The following section discusses these topics.

4.3.1 Changing Terminals

If you ever need to log in to XENIX on a terminal of a type different than the terminal you normally use, you may need to change the shell TERM variable. This is normally set to the proper terminal when you log in, but if you switch terminal types you will have to reset the TERM variable. To reset this variable, type the following line at command level:

TERM=*termname*

where *termname* is the name of a known terminal. A list of known terminals is described in *terminals*(M). A variety of terminals are supported; terminal capabilities are listed in the system file */etc/termcap*.

4.3.2 Setting Terminal Options

There are a number of terminal options that can be set with the command **stty**. When entered without parameters, **stty** displays the current terminal settings. For example, typical output might look like this:

```
speed 9600 baud
erase '^h' ; kill '^u'
even -nl
```

Each of the above characteristics can be set with **stty**. For more information, see *stty*(C) in the *XENIX Reference Manual*.

4.4 Editing the Command Line

When you sit in front of a terminal and type commands at your keyboard, there are a number of special keys that you can use. The most useful ones are described below.

4.4.1 Entering a Command Line

From your terminal, entering a command line consists of typing characters then pressing RETURN. Once you have pressed RETURN the computer reads the command line and commands specified on that line are executed. You may type as many command lines as you want without waiting for commands to complete, because XENIX supports type-ahead of characters.

4.4.2 Erasing a Command Line

When entering commands, typing errors are bound to occur. To erase the current command line, press CNTRL-U.

4.4.3 Halting Screen Output

In many cases, you will be examining the contents of a file on the terminal screen. For longer files, the contents will often scroll off the screen faster than you can examine them. To temporarily halt a program's output to the terminal screen, press CNTRL-S. To resume output, press CNTRL-Q.

4.5 Manipulating Files

File manipulation (creating, displaying, combining, copying, moving, naming, and deleting files), is one of the most important capabilities an operating system provides. The XENIX commands that perform these functions are described in the following sections.

4.5.1 Creating a File

To create a file and place text in it, use the editor **vi**, described in Chapter 5 of this manual, "Vi: A Text Editor". If for some reason you wish to create an empty file, type

```
> filename
```

Where *filename* is the name of the empty file. In general, new files are created by commands *as needed*.

4.5.2 Displaying File Contents

The **more** command displays the contents of a file one screenful at a time. It has the form

```
more options filename
```

More is useful for looking at a file when you don't want to make changes to it. For example, to display the contents of the file *memos*, type

```
more memos
```

More can be invoked with options that control where the display begins, and how the file is displayed. These options include:

+linenumber

Begins the display at the line in the file designated by *linenumber*.

+/text

Begins the display two lines before *text*, where *text* is a word or number. If *text* is two or more words, they must be enclosed in double quotation marks.

-c Redraws the screen instead of scrolling.

-r Displays control characters, which are normally ignored by **more**.

To begin looking at the file *memo* at the first occurrence of the words "net gain", for example, type

```
more +/"net gain" memo
```

If the file is more than one screenful long, the percentage of the file that remains is displayed on the bottom line of the screen. To look at more of the file, use the following scrolling commands:

RETURN Scrolls down one line.

d Scrolls down one-half screen.

SPACE Scrolls down a full screen.

nSPACE Scrolls down *n* lines.

. Repeats the previous command.

You cannot scroll backward, toward the beginning of the file.

You can search forward for patterns in **more** with the slash (/) command. For example, to search for the pattern "net gain", type

```
/net gain/
```

and press RETURN. **More** displays the message

skipping...

at the top of the screen, and scrolls to a location two lines above ‘net gain’.

If you are looking at a file with **more** and decide you want to change the file, you can invoke the **vi** editor by pressing

v

See Chapter 5, “Vi: A Text Editor” for information on using **vi**.

More quits automatically when it reaches the end of a file. To exit **more** before the end of a file, type

q

The **head** and **tail** commands display the first and last ten lines of a file, respectively. They are useful for checking the contents of a particular file.

For example, to look at the first ten lines of the file *memo*, type

head memo

You can also specify how many lines the **head** and **tail** commands display. For example,

tail -4 memo

displays the last four lines of *memo*.

The **cat** command also displays the contents of a file. **Cat** scrolls the file until you press CNTRL-S to stop it. Pressing CNTRL-Q will continue the scrolling. **Cat** stops automatically at the end of a file. If you wish to stop the display before the end of the file, press INTERRUPT. To display the contents of one file, type

cat file1

To display the contents of more than one file, type

cat file1 file2 file3

4.5.3 Combining Files

The **cat** command is frequently used to combine files into some other new file. Thus, to combine the two files named *file1* and *file2*, into a new file named *bigfile*, type:

cat file1 file2 > bigfile

Note here that we are putting the contents of the two files into a new file with the name *bigfile*. The greater than sign (**>**) is used to *redirect* output of the **cat** command to the new file.

You can also use **cat** to append one file to the end of another file. For example, to append *file1* to *file2*, type

cat file1 >> file2

The contents of *file1* are added to *file2*. *File1* still exists as a separate entity.

4.5.4 Moving a File

The **mv** command moves a file into another file in the same directory, or into another directory. For instance, to move a file named *text* to a new file named *book*, type:

mv text book

After this move is completed, no file named *text* will exist in the working directory, because the

file has been renamed *book*. To move a file into another directory, give the name of the destination directory as the final name in the **mv** command. For instance, to move *file1* and *file2* into the directory named */tmp*, type:

```
mv file1 file2 /tmp
```

The two files you have moved no longer exist in your working directory, but now exist in the directory */tmp*. The above command has exactly the same effect as typing the following two commands:

```
mv file1 /tmp/file1
mv file2 /tmp/file2
```

The **mv** command always checks to see if the last argument is the name of a directory and, if so, all files designated by filename arguments are moved into that directory.

4.5.5 Renaming a File

To rename a file, simply “move” it to a file with the new name: the old name of the file is removed. Thus, to rename the file *anon* to *johndoe*, type:

```
mv anon johndoe
```

Note that moving and renaming a file are essentially identical operations.

4.5.6 Copying a File

There are two forms of the **cp** command: one in which files are copied into a directory, and another in which a file is copied to another file. Thus, to copy three files into a directory named *filedir*, type:

```
cp file1 file2 file3 filedir
```

In the above command, three files are copied into the directory *filedir*; the original versions still reside in the working directory. Note that the filenames are identical in the two directories. Like the **mv** command, **cp** always checks to see if the last argument is the name of a directory, and, if so, all files designated by filename arguments are copied into that directory.

To create two copies of a file in your own working directory, you must rename the copy. To do this, the copy command can be invoked as follows:

```
cp file filecopy
```

After the above command has executed, two files with identical contents reside in the working directory. To learn how to copy directories, see section 4.6.7, “Copying a Directory”, later in this chapter.

4.5.7 Deleting a File

To delete or remove files, type:

```
rm file1 file2
```

In the above command, the files *file1* and *file2* are removed from your working directory. The command

```
rm -i file1 file2
```

allows you to interactively remove files by asking you if you really want to delete each of the files *file1* and *file2*. If you press *y* followed by a RETURN, the given file is removed; if you press *n* the file is left untouched. This command is useful when cleaning up a directory that contains many

files.

4.5.8 Finding Files

The **find** command searches for files that have a specified name. **Find** is useful for locating files that have the same name, or just for finding a file if you have forgotten which directory it is in. The command has the form:

```
find pathname -name filename -print
```

The *pathname* is the pathname of the directory you want to search. **Find** searches recursively, that is, it starts at the named directory and searches downward through all files and subdirectories under the directory specified in *pathname*.

The “-name” option indicates that you are searching for files that have a specific *filename*. (There are other search conditions you can use with **find**; see *find(C)* in the *XENIX Reference Manual*.)

Filename is the name of the file you are searching for.

The “-print” option indicates you want to print the pathnames of all the files that match *filename* on your terminal screen. You may direct this output to a file instead of your screen with the output redirection symbol, **>**. (There are other actions that can be performed with **find**, such as removing and moving files; see *find(C)* in the *XENIX Reference Manual*.)

For example, the following command finds every file named *memo* in the directory */usr/joe* and all its subdirectories:

```
find /usr/joe -name memo -print
```

The output might look like this:

```
/usr/joe/memo
/usr/joe/accounts/memo
/usr/joe/meetings/memo
/usr/joe/mail/memo
```

4.5.9 Linking a File to Another File

The **ln** command joins two files in different directories so that when the file is changed in one directory, it is also changed in the other directory. This can be useful if several users need to share information, or if you want a file to appear in more than one directory. This command has the form

```
ln file newfile
```

where *file* is the original file, and *newfile* is the new, linked file. For example, the following command links *memos* in */usr/joe* to *joememos* in */usr/mary*:

```
ln /usr/joe/memos /usr/mary/joememos
```

Whenever */usr/joe/memos* is updated, the file */usr/mary/joememos* is also changed.

When you link files a name is associated with an *inode*. An *inode* specifies a unique set of data on the disk. One or more names can be associated with this data. Thus, the above command assures that the files *dir1/file1* and *dir2/file2* have identical contents.

There are three things to remember about linking files that are not immediately obvious:

1. Linking large sets of files to other parallel files can save a considerable amount of disk space.

2. Linking files used by more than one person is risky, because any party can alter the file and thus affect the contents of all files linked to it.
3. Removing a file from a directory does not remove other links to the file. Thus the file is not truly deleted from the system. For example, if you delete a file that has 4 links, 3 links remain.

For more information about linking see *ln(C)* in the *XENIX Reference Manual*.

4.6 Manipulating Directories

Because of the hierarchical organization of the file system, there are many directories and subdirectories in the XENIX system. Within the file system are directories for each user of the system. Within your user directory you can create, delete, and copy directories. Commands that let you manipulate directories are described in the following sections.

4.6.1 Printing the Name of Your Working Directory

All commands are executed relative to a “working” directory. The name of this directory is given by the **pwd** command, which stands for “print working directory”. For instance, if your current working directory is */usr/joe*, when you type:

```
pwd
```

you will get the output:

```
/usr/joe
```

You should always think of yourself as residing “in” your working directory.

4.6.2 Listing Directory Contents

You can list the contents of a directory with the **lc** command. This command sorts and lists the names of files and directories in a given directory in columns. If no directory name is given, **lc** lists the contents of the current directory. The **lc** command has the form

```
lc options name
```

For example, to list the contents of the directory *work*, type

```
lc work
```

Your output might look like this:

```
accounts  meetings  notes
mail      memos     todo
```

If no *name* is specified, **lc** lists the contents of the current directory. If *accounts* is the current directory, for example, the command

```
lc
```

lists the names of the files and subdirectories in that directory.

The following options control the sort order and the information displayed by the **lc** command:

- a Lists all files in the directory, including the “hidden” files (filenames that begin with a dot, such as *.profile* and *.mailrc*).
- r Lists names in reverse alphabetical order.

- t Lists names in order of last modification, the latest (most recently modified) first. When used with the **-r** option, lists the oldest first.
- R Lists all files and directories in the current directory, plus each file and directory *below* the current one. The “R” stands for “recursive”.
- F Marks directories with a backslash(\) and executable files with an asterisk (*).

The **l** command gives a “long” listing of a directory, producing an output that might look something like this:

```
total 501
drwxr-x--- 2 boris   grp1    272 Apr  5 14:33 dir1
drwxr-x--- 2 enid    grp1    272 Apr  5 14:33 dir2
drwxr-x--- 2 iris    grp1    592 Apr  6 11:12 dir3
-rw-r----- 1 olaf    grp2    282 Apr  7 15:11 file1
-rw-r----- 1 olaf    grp2     72 Apr  7 13:50 file2
-rw-r----- 1 olaf    grp2   1403 Apr  1 13:22 file3
```

Reading from left to right, the information given for each file or directory includes:

- Permissions
- Number of links
- Owner
- Group
- Size in bytes
- Time of last modification
- Filename

The information in this listing and how to change permissions are discussed below in Section 4.8, “Using File and Directory Permissions”.

The **l** command takes the same options as **lc**.

For more information about listing the contents of a directory, see *ls(C)* in the XENIX *Reference Manual*.

4.6.3 Creating a Directory

To create a subdirectory in your working directory, use the **mkdir** command. For instance, to create a new directory named *phonenumbers*, simply type:

```
mkdir phonenumbers
```

After this command has been executed, a new empty directory will exist in your home directory.

4.6.4 Removing a Directory

To remove a directory located in your working directory, use the **rmdir** command. For instance, to remove the directory named *phonenumbers* from the current directory, simply type:

```
rmdir phonenumbers
```

Note that the directory *phonenumbers* must be *empty* before it can be removed; this prevents

catastrophic deletions of files and directories. If you want to live dangerously, it is possible to recursively remove the contents of a directory using the **rm** command, but that will not be explained here. See *rm(C)* in the *XENIX Reference Manual* for more information.

4.6.5 Renaming a Directory

To rename a directory, use the **mv** command. For instance, to rename the directory *little.dir* to *big.dir*, type:

```
mv little.dir big.dir
```

This is a simple renaming operation; no files are moved.

4.6.6 Moving a Directory

The **mv** command also moves directories. This command has the form

```
mv olddirectory newdirectory
```

where *Newdirectory* is a directory that already exists. For example, to move the directory */usr/joe/accounts* into */usr/joe/overdue* type

```
mv /usr/joe/accounts /usr/joe/overdue
```

The new pathname of */usr/joe/accounts* is */usr/joe/overdue/accounts*.

4.6.7 Copying a Directory

The **copy** command copies directories. This command has the form

```
copy options olddir newdir
```

To copy all the files in the directory */usr/joe/memos* into */usr/joe/notes* type

```
copy /usr/joe/memos /usr/joe/notes
```

The files in */usr/joe/memos* are copied into */usr/joe/notes*. The **copy** command has the following options:

- l Links the copied files to the original.
- m Gives the copied files the same modification dates as the original files.
- r Copies the directory recursively, i.e., copies all the directories under the named directory.

4.7 Moving in the File System

When using the XENIX system, it helps to imagine a large tree structure of files and directories. Each directory should be thought of as a place that you can move into or out of. At all times you are “someplace” in the tree structure. This place is called either your working directory or current directory. The commands used to find out where you are and to move around in the tree structure are discussed below.

4.7.1 Finding Out Where You Are

Your current location in the file system is the name of the working directory. You can find out this name by using the **pwd** command, which stands for “print working directory”. For example, if

you are in the directory */usr* then typing the command

```
pwd
```

prints out the name:

```
/usr
```

4.7.2 Changing Your Working Directory

Your working directory represents your location in the file system: it is “where you are” in XENIX. To alter this location in the XENIX file system, use the change directory (**cd**) command:

```
cd
```

This changes your working directory to your home directory. To move to any other directory, specify that directory as an argument to **cd**. For instance, the following command:

```
cd /usr
```

moves you to the */usr* directory. Because you are always “in” your working directory, changing working directories is much like “traveling” from directory to directory.

To move up one directory from your current directory, type

```
cd ..
```

For example, the above command would move you from the directory */usr/joe/work* to */usr/joe*. Similarly, the command

```
cd ../..
```

would move you from the directory */usr/joe/work* to */usr*, moving you up *two* directories.

4.8 Using File and Directory Permissions

The XENIX system allows the owner to restrict access to files and directories, limiting who can read, write and execute files owned by him. To determine the permissions associated with a given file or directory, use the **l** command. The output from the **l** command should look something like this:

```
total 501
drwxr-x--- 2 boris  grp1  272 Apr  5 14:33 dir1
drwxr-x--- 2 enid   grp1  272 Apr  5 14:33 dir2
drwxr-x--- 2 iris   grp1  592 Apr  6 11:12 dir3
-rw-r----- 1 olaf   grp2  282 Apr  7 15:11 file1
-rw-r----- 1 olaf   grp2   72 Apr  7 13:50 file2
-rw-r----- 1 olaf   grp2 1403 Apr  1 13:22 file3
```

first file in the above list, are

```
drwxr-x---
```

The first character indicates the type of file and must be one of the following:

- Indicates an ordinary file.
- d Indicates a directory.
- c Indicates a character special device such as a lineprinter or terminal.
- b Indicates a block special device such as a hard or floppy disk.

- n Indicates a name special file (i.e., a semaphore used for controlling access to some resource).
- s Indicates a shared data file.
- p Indicates a named pipe.

From left to right, the next nine characters are interpreted as three sets of three permissions each. Each respective set of three indicates the following permissions:

- Owner permissions
- Group permissions
- All other user permissions

Within each set, the three characters indicate permission to read, to write, and to execute the file as a command, respectively. For a directory, "execute" permission means permission to search the directory for any included files or directories.

Ordinary file permissions have the following meanings:

- r The file is readable.
- w The file is writeable.
- x The file is executable.
- The indicated permission is not granted.

For directories, permissions have the following meanings:

- r Files can be listed in the directory; the directory must also have "x" permission.
- w Files can be created or deleted in the directory; as with "r", the directory itself must also have "x" permission.
- x The directory can be searched. A directory must have "x" permission before you can move to it with the `cd` command (i.e., `cd` to it), access a file within it, or list the files in it. Remember that a user must have "x" permission *to do anything useful to the directory*.

The following are some typical directory permission combinations:

- d----- No access at all. This is the mode that denies access to the directory to a class of users.
- drwx----- Allows access by the owner to use `lc`, create files, delete files, access files (subject to file permissions), and `cd` to the directory. This is the typical permission for the owner of a directory.
- drwxr-x--- Allows access by members of the group to use `lc`, and access files subject to file permissions. Group members can `cd` to this directory, but cannot create or delete files in it. This is the typical permission an owner gives to others who need access to files in his directory.
- drwx--x--x With these permission settings users other than the owner cannot use `lc` but can `cd` to the directory. Other users can only access a file within this directory by its exact name; they cannot use special characters. Files

cannot be created or deleted in the directory by anyone except the owner. This mode is rarely used, but can be useful if you want to give someone access to a specific file in a directory without permitting access to other files in the same directory.

This chapter discusses ordinary files, executable files, and directories only. For information about other types of files, see `ls(C)` in the *XENIX Reference Manual*.

4.8.1 Changing Permissions

The **chmod** command changes the read, write, execute, and search permissions of a file or directory. This command is useful if you have created a file in one mode, but want to give others permission to read, write or execute it. The **chmod** command has the form

```
chmod instruction filename
```

The *instruction* segment of the command indicates which permissions you want to change for which class of users. There are three classes of users, and they are indicated as follows:

- u User, the owner of the file or directory
- g Group, the group the owner of the file belongs to
- o Other, all users of the system
- a All classes of users

There are three types of permissions, as follows:

- r Read, which allows permitted users to look at but not change or delete the file.
- w Write, which allows permitted users to change or even delete the file.
- x Execute, which allows permitted users to execute the file as a command.

For example, assume *file1* exists with the following permissions:

```
-rw-r-----
```

In the above example, the owner of the file has read and write permission, group members have read permission, and others have no access at all.

To give *file1* read permission for *all* classes of users, type:

```
chmod a+r file1
```

In the instruction segment of the command (*a+r*) the “a” stands for “all”. The resulting permissions are:

```
-rw-r--r--
```

For *file1* with the attributes

```
-rw-----
```

The following command gives write and execute permissions to members of a group only:

```
chmod g+wx file1
```

This command would alter the permission attributes so they look like this:

```
-rw--wx---
```

To remove write and execute permission by the user (owner) and group associated with *file1*, type:


```
chmod ug-wx file1
```

4.8.2 Changing Directory Search Permissions

Directories also have an execute permission. This attribute signifies search permission, rather than execute permission, since directories cannot be executed. If this permission is denied to a particular user, then that user cannot even list the names of the files in the directory.

For example, assume that the directory *dir1* has the following attributes:

```
drwxr-xr-x
```

To remove search permission for other users to examine *dir1*, type:

```
chmod o-xr dir1
```

The new attributes for *dir1* are:

```
drwxr-x---
```

4.9 Processing Information

In many cases, files will contain information that you may want to process. Various utility programs exist on XENIX to process information. A set of these programs and their uses are described in the following sections.

4.9.1 Comparing Files

To compare two text files use the **diff** command to print out those lines that differ between the files that you specify. For example, suppose that a file named *men* has the contents

```
Now is the time for all good men to  
Come to the aid of their party.
```

and that a file named *women* has the following contents:

```
Now is the time for all good women to  
Come to the aid of their party.
```

If this is the case, then the command

```
diff men women
```

produces the following results:

```
1c1  
< Now is the time for all good men to  
---  
> Now is the time for all good women to
```

A three-way difference listing can be created with the **diff3** command. For information about **diff3** see *diff3(C)* in the XENIX *Reference Manual*.

4.9.2 Echoing Arguments

The **echo** command echos arguments to the standard output. For example, typing:

```
echo hello
```

outputs:

hello

on the terminal screen. To output several lines of text, surround the echoed argument in double quotation marks and press RETURN between lines. A secondary prompt will appear until you type the final double quotation mark. For example, type:

```
echo "Now is the time
For all good men
To come to the
Aid of their party."
```

This will output:

```
Now is the time
For all good men
To come to the
Aid of their party.
```

Echo is particularly useful if you should ever program in the shell command language. For more information about the shell, see Chapter 7, "The Shell".

4.9.3 Sorting a File

One of the most useful file processing commands is **sort**. By default, **sort** sorts the lines of a file according to the ASCII collating sequence (i.e., it alphabetizes them). For example, to sort a file named *phonelist*, type:

```
sort phonelist
```

In the above case, the sorted contents of the file are displayed on the screen. To create a sorted version of *phonelist* named *phonesort*, type:

```
sort phonelist > phonesort
```

Note that **sort** is useful for sorting the output from other commands. For example, to sort the output from execution of a **who** command, type:

```
who | sort > whosort
```

This command takes the output from **who**, sorts it, and then sends the sorted output to the file *whosort*.

A wide variety of options are available for **sort**. For more information, see *sort* (C) in the XENIX Reference Manual.

4.9.4 Searching for a Pattern in a File

The **grep** command selects and extracts lines from a file, printing only those lines that match a given pattern. For example, to print out all lines in a file containing the word "tty38", type:

```
grep 'tty38' file
```

In general, you should always enclose the pattern you are searching for in single quotation marks ('), so that special metacharacters are not expanded unexpectedly by the shell.

As another example, assume that you have a file named *phonelist* that contains a name followed by a phone number on each line. Assume also that there are several thousand lines in this list. You can use **grep** to find the phone number of someone named Joe, whose phone number prefix is 822, as follows:

```
grep 'joe' phonelist | grep '822-' > joes.number
```

Grep finds all occurrences of lines containing the word "joe" in the file *phonelist*. The output

from this command is then filtered through another **grep** command, which searches for an "822-" prefix, thus removing any unwanted joes. Finally, assuming that a unique phone number for joe exists with the "822-" prefix, that name and number are placed in the file *joes.number*.

For more information about **grep**, its relatives **fgrep** and **egrep**, and the types of patterns it can be used to search for (called regular expressions) see *grep* (C) in the XENIX *Reference Manual*.

4.9.5 Counting Words, Lines, and Characters

Wc is a utility for counting words in a file. The letters "wc" stand for word count. Words are presumed to be separated by punctuation, spaces, tabs, or newlines. **Wc** also counts characters and lines; all three counts are reported by default. For example, to count the number of lines, words, and characters in the file *textfile*, type:

```
wc textfile
```

Typical output describing lines, words and characters might be:

```
4432 18188 97808 textfile
```

To specify a count of characters, words, or lines only, you must use an appropriate mnemonic switch. To illustrate, examine the following three commands and the output produced by each:

```
wc -c textfile
97808 textfile
```

```
wc -w textfile
18188 textfile
```

```
wc -l textfile
4432 textfile
```

The first example prints out the number of characters in *textfile*, the second prints out the number of words, and the third prints out the number of lines.

4.9.6 Delaying a Process

The **at** program allows you to set up commands to be executed at a specified time. It is useful if you want to execute a command when you are not planning to be at your terminal, or even logged in.

The **at** command has the form

```
at time day file
```

Time is the time of day, in digits, followed by "am" or "pm". One- and two-digit numbers are interpreted as hours, three- and four-digit numbers as hours and minutes. More than four digits is not permitted.

Day is optional. It is either a month name followed by a day number, or a day of the week. If no day is specified, the command will be executed today.

File is the name of the file that contains the command or commands to be executed.

For example, if you want to find out what processes are running at 10 pm on Tuesday, place the following line in a file named *use*

```
ps a > /usr/myname/use
```

(See Chapter 6, "Vi: A Text Editor", for information on creating and inserting text into files.)

After you have written out the file and returned to command level, type

at 10pm tues use

Press RETURN. The XENIX prompt reappears and you may continue working. At 10 pm on Tuesday, XENIX will execute **psa** and place the output in the file *use*. **At** is unaffected by logging out.

To check what files you have waiting to be processed, use the **atq** command. **Atq** lists the files to be processed, along with the following information:

- The file's user ID
- The file's ID number
- The date and time the file will be processed

To cancel an **at** command, first check the list of files to be processed and note the file ID number. Then use the **atrm** command to remove the file or files from the list. The **atrm** command has the form:

atrm *number*

For example,

atrm 84.032.2300.21

removes file number 84.032.2300.21, canceling whatever commands were included in that file. A user can only remove his own files.

4.10 Controlling Processes

In XENIX, several processes can run at the same time. For example, you may run the **sort** program on a file in the "background", and edit another file in the "foreground" while the **sort** program is running. Things that you directly control at your keyboard are called "foreground" processes. Other processes, which you can initiate but that you otherwise have little control over, are called background processes. At any one time you can have only one foreground process executing, but multiple background processes may execute simultaneously. Controlling foreground and background processes is the subject of this section.

4.10.1 Placing a Process in the Background

Normally, commands sent from the keyboard are executed in strict sequence; one command must finish executing before the next can begin. Executing commands of this type are called foreground processes. A background process, in contrast, need not finish executing before you give your next command. Background commands are especially useful for commands that may take a long time to complete.

To place a process in the background, type an ampersand (&) at the end of the command. For example, to count the number of words in several large files while simultaneously continuing with whatever else you have to do, type:

wc file1 file2 file3 >count&

Output is collected in the file *count*. If output were not put in *count*, it would appear on the screen at unpredictable times as you continue with your work.

When processes are placed in the background, you lose control of them as they execute. For instance, typing INTERRUPT does *not* abort a background process. You must use the **kill**

command, described in the following section, instead.

4.10.2 Killing a Process

To stop execution of a foreground process, press your terminal's INTERRUPT key. This kills whatever foreground command is currently running. To kill all your processes executing in the background, type:

```
kill 0
```

To kill only a specified process executing in the background, first type:

```
ps
```

Ps displays the Process Identification Numbers (PIDs) of your existing processes:

PID	TTY	TIME	CMD
3459	03	0:15	-s h
4831	03	1:52	cc program.s
5185	03	0:00	ps

Next, you might type

```
kill 4831
```

where 4831 is the PID of the process that you want killed.

Note

Killing a process associated with the **vi** editor may leave the terminal in a strange mode. Also, temporary files that are normally created when a command starts, and are deleted when the command finishes, may be left behind after a **kill** command. Temporary files are normally kept in the directory */tmp*. This directory should be checked periodically and old files deleted.

4.11 Getting Status Information

Because XENIX is a large, self-contained computing environment, there are many things that you may want to find out about the system itself, such as who is logged in, how much disk space there is, what processes are currently running. This section explains the types of information available from the system and how to get it.

4.11.1 Finding Out Who is on the System

The **who** command lists the names, terminal line numbers, and login times of all users currently logged on to the system. For example, type:

```
who
```

This command produces something like the following output on your terminal screen:

```

arnold    tty02    Apr   7 10:02
daphne    tty21    Apr   7 07:47
elliott   tty23    Apr   7 14:21
ellen     tty25    Apr   7 08:36
gus       tty26    Apr   7 09:55
adrian    tty28    Apr   7 14:21

```

The **finger** command provides more detailed information, such as office numbers and phone extensions. For more information, about using **finger** see *finger(C)* in the *XENIX Reference Manual*.

4.11.2 Finding Out What Processes Are Running

Because commands can be placed in the background for processing, it is not always obvious which processes you are responsible for. The **ps** command stands for “process status” and displays information about currently running processes associated with your terminal. For instance, the output from a **ps** command might look like this:

```

PID TTY TIME CMD
10308 38 1:36 ed chap02.man
    49 38 0:29 -sh
11267 38 0:00 ps

```

The PID column gives a unique process identification number that can be used to kill a particular process. The TTY column shows the terminal that the process is associated with. The TIME column shows the cumulative execution time for the process. Processes can be killed using the **kill** command. See section 4.10.2, “Killing a Process” for information on how to use the **kill** command.

To find out all the processes running on the system, use the **a** option:

```
ps a
```

To find out about the processes running on a terminal other than the terminal you are using, specify the terminal number. For example, to find out what processes are associated with terminal 13, type:

```
ps t13
```

For more information about **ps** and its options, see *ps(C)* in the *XENIX Reference Manual*.

4.11.3 Getting Lineprinter Information

At times it may be necessary to know how many files are queued up at the lineprinter. This information can be found by listing the directory in which queued files reside, */usr/spool/lpd*. To examine this directory, type:

```
ls -l /usr/spool/lpd
```

4.12 Using the Lineprinter

The following sections describe the commands that will help you use your lineprinter effectively and efficiently.

4.12.1 Sending a File to the Lineprinter

One of the most common operations that you will want to perform is printing files on the lineprinter. The most straightforward method for doing this is to type


```
lpr file1
```

for one file, or:

```
lpr file1 file2 file3
```

for multiple files. Other common uses of **lpr** involve pipes. For instance, to paginate and print a file of raw text, type:

```
pr textfile | lpr
```

The **pr** and **lpr** commands are very often used together. As another example, to sort, paginate, and print a file, type:

```
sort datafile | pr | lpr
```

4.12.2 Getting Lineprinter Queue Information

XENIX does not require that a file be entirely printed before the **lpr** command finishes. Instead, **lpr** makes sure only that the file is placed in a special directory where it will wait its turn to be printed.

The files in this queue are contained in the directory */usr/spool/lpd*. To examine this lineprinter queue, type:

```
ls -l /usr/spool/lpd
```

4.13 Communicating with Other Users

Because the XENIX system supports multiple users, it is very convenient to communicate with other users of the system. The various methods of communication are described below.

4.13.1 Sending Mail

Mail is a system-wide facility that permits you and other system users to send and receive mail. To send mail to another user on the system, type:

```
mail joe
```

where *joe* is the name of any user of the system. Following entry of the command, you enter the actual text of the message you want to send. Entry of text is terminated by typing a CNTRL-D.

A complete session at the terminal might look like this on your screen:

```
mail -s "Meeting today" joe
There will be a meeting at 2:00 today
to review recent problems with the
new system.
CNTRL-D
```

Note the use of the **-s** switch to specify the subject of the message.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to yourself is a handy reminder mechanism.) You can also send a previously prepared letter, and you can send mail to a number of people all at once. For more details see Chapter 6, "Mail", and *mail*(C) in the *XENIX Reference Manual*.

4.13.2 Receiving Mail

When you log in, you may sometimes get the message:

you have mail

To read your mail, type:

mail

A heading for each message is then displayed on your terminal screen. When you press RETURN, the contents of the first message are displayed. Subsequent messages are displayed, one message at a time, most recent message first, each time you press RETURN.

After each message is displayed, **mail** waits for you to tell it what to do with the message. The two basic responses are **d**, which deletes the message, and RETURN, which does not delete the message (so it will still be there the next time you read your mailbox). To exit mail, type **q**, for "quit". Other responses are described in the *XENIX Reference Manual* under *mail (C)*.

4.13.3 Writing to a Terminal

To write directly to another user's terminal, use the **write** command. For example, to write to joe's terminal, type:

write joe

After you have executed the command by pressing RETURN, each subsequent line that you type is displayed both on your own terminal screen and on joe's. To terminate the writing of text to joe, enter a CNTRL-D alone on a line. The procedure for a two-way write is for each party to end each message with a distinctive signal, normally (o) for "over"; when a conversation is about to be terminated use the signal (oo) for "over and out".

4.14 Using the System Clock and Calendar

There are several XENIX commands that will tell you the date and time, or display a calendar for any month or year you choose. The following sections explain these commands.

4.14.1 Finding Out the Date and Time

The *date* command displays the time and date. Type

date

The date and time are displayed in the bottom left corner of the screen.

4.14.2 Displaying a Calendar

The **cal** command displays the calendar of any month or year you specify. This command has the form:

cal *month year*

For example, to display the calendar for March, 1952 type

cal 3 1952

The result is:


```
      March 1952
S   M Tu   W Th   F   S
      1
2   3   4   5   6   7   8
9  10  11  12  13  14  15
16 17  18  19  20  21  22
23 24  25  26  27  28  29
30 31
```

The month must always be expressed as a digit. To display the calendar for an entire year, leave out the month. The year must always be expressed in full; the command "cal 84" displays the calendar for the year 84, not 1984.

4.15 Using the Automatic Reminder Service

An automatic reminder service is normally available for all XENIX users. Once each day, XENIX uses the **calendar** command to examine each user's home directory for a file named *calendar*, the contents of which might look something like this:

```
1/23 David's wedding
2/9  Mira's birthday
3/30 Paul's birthday
4/27 Meeting at 2:00
9/1  Karen's birthday
10/3 License renewal
```

Calendar examines each line of the calendar file, extracting from the file those lines containing today's and tomorrow's dates. These lines are then mailed to you to remind you of the specified events.

4.16 Using Another User's Account

You can easily access another user's files, regardless of the permission settings, with the **su** command. The **su** procedure resembles logging in, and you must know the other user's password. For example, to become user Joe, type

```
su joe
```

and press RETURN. When the password prompt appears, type Joe's password. To cancel the effect of the **su** command and return to your own account, press CNTRL-D.

4.17 Calculating

The **bc** command invokes an interactive desk calculator that can be used as if it were a hand-held calculator. A typical session with **bc** is shown below. Comments explain what action is performed after each input line:

```

/* This is a comment */
123.456789 + 987.654321 /* Add and output */
1111.111110
9.0000000 - 9.0000001 /* Subtract and output */
-.0000001
64/8 /* Divide and output */
8
1.12345678934 * 2.3 /* Note precision */
2.58395061548
19%4 /* Find remainder */
3
3^4 /* Exponentiation */
81
2/1*2 /* Note precedence */
4
2/(1*2) /* Note precedence again */
1
x = 46.5 /* Assign value to x */
y = 52.5 /* Assign value to y */
x + y + 1.0000 /* Add and output */
100.0000
obase=16 /* Set hex output base */
15 /* Convert to hex */
F
16 /* Convert to hex */
10
64 /* Convert to hex */
40
255 /* Convert to hex */
FF
256 /* Convert to hex */
100
512 /* Convert to hex */
200
quit /* Must type whole word */

```

Also available are scaling, function definition, and programming statements much like those in the C programming language. Other features include assignment to named registers and subroutine calling. For more information, see Chapter 8, "BC: A Calculator".

Chapter 5

Vi: A Text Editor

5.1 Introduction	1
5.2 Demonstration	1
5.2.1 Entering the Editor	1
5.2.2 Inserting Text	2
5.2.3 Repeating a Command	3
5.2.4 Undoing a Command	3
5.2.5 Moving the Cursor	4
5.2.6 Deleting	5
5.2.7 Searching for a Pattern	7
5.2.8 Searching and Replacing	8
5.2.9 Leaving Vi	10
5.2.10 Adding Text From Another File	10
5.2.11 Leaving Vi Temporarily	11
5.2.12 Changing Your Display	11
5.2.13 Canceling an Editing Session	12
5.3 Editing Tasks	12
5.3.1 How to Enter the Editor	12
5.3.2 Moving the Cursor	13
5.3.3 Moving Around in a File: Scrolling	15
5.3.4 Inserting Text Before the Cursor: i and I	15
5.3.5 Appending After the Cursor: a and A	16
5.3.6 Correcting Typing Mistakes	16
5.3.7 Opening a New Line	16
5.3.8 Repeating the Last Insertion	16
5.3.9 Inserting Text From Other Files	17
5.3.10 Inserting Control Characters into Text	19
5.3.11 Joining and Breaking Lines	20
5.3.12 Deleting a Character: x and X	20
5.3.13 Deleting a Word: dw	20
5.3.14 Deleting a Line: D and dd	20
5.3.15 Deleting an Entire Insertion	21
5.3.16 Deleting and Replacing Text	21
5.3.17 Moving Text	23
5.3.18 Searching: / and ?	27
5.3.19 Searching and Replacing	27
5.3.20 Pattern Matching	29
5.3.21 Undoing a Command: u	30
5.3.22 Repeating a Command: .	32
5.3.23 Leaving the Editor	32
5.3.24 Editing a Series of Files	33
5.3.25 Editing a New File Without Leaving the Editor	34
5.3.26 Leaving the Editor Temporarily: Shell Escapes	34
5.3.27 Performing a Series of Line-Oriented Commands: Q	35
5.3.28 Finding Out What File You're In	35
5.3.29 Finding Out What Line You're On	36

5.4 Solving Common Problems 36

5.5 Setting Up Your Environment 37

- 5.5.1 Setting the Terminal Type 37
- 5.5.2 Setting Options: The set Command 38
- 5.5.3 Displaying Tabs and End-of-Line: list 39
- 5.5.4 Ignoring Case in Search Commands: ignorecase 39
- 5.5.5 Displaying Line Numbers: number 39
- 5.5.6 Printing the Number of Lines Changed: report 39
- 5.5.7 Changing the Terminal Type: term 39
- 5.5.8 Shortening Error Messages: terse 39
- 5.5.9 Turning Off Warnings: warn 40
- 5.5.10 Permitting Special Characters in Searches: nomagic 40
- 5.5.11 Limiting Searches: wrapscan 40
- 5.5.12 Turning on Messages: msg 40
- 5.5.13 Customizing Your Environment: The .exrc File 40

5.6 Summary of Commands 42

5.1 Introduction

Any ASCII text file, such as a program or document, may be created and modified using a text editor. There are two text editors available on the XENIX system, **ed** and **vi**. **Ed** is discussed in Appendix A of this manual.

Vi (which stands for “visual”) combines line-oriented and screen-oriented features into a powerful set of text editing operations that will satisfy any text editing need.

The first part of this chapter is a demonstration that gives you some hands-on experience with vi. It introduces the basic concepts you must be familiar with before you can really learn to use vi, and shows you how to perform simple editing functions. The second part is a reference that shows you how to perform specific editing tasks. The third part describes how to set up your vi environment and how to set optional features. The fourth part is a summary of commands.

Because vi is such a powerful editor, it has many more commands than you can learn at one sitting. If you have not used a text editor before, the best approach is to become thoroughly comfortable with the concepts and operations presented in the demonstration section, then refer to the second part for specific tasks you need to perform. All the steps needed to perform a given task are explained in each section, so some information is repeated several times. When you are familiar with the basic vi commands you can easily learn how to use the more advanced features.

If you have used a text editor before, you may want to turn directly to the task-oriented part of this chapter. Begin by learning the features you will use most often. If you are an experienced user of vi you may prefer to use *Vi(C)* in the *XENIX Reference Manual* instead of this chapter.

This chapter covers the basic text editing features of vi. For more advanced topics, and features related to editing programs, refer to *Vi(C)* in the *XENIX Reference Manual*.

5.2 Demonstration

The following demonstration gives you hands-on experience using vi, and introduces some basic concepts that you must understand before you can learn more advanced features. You will learn how to enter and exit the editor, insert and delete text, search for patterns and replace them, and how to insert text from other files. This demonstration should take one hour. Remember that the best way to learn vi is to actually use it, so don't be afraid to experiment.

Before you start the demonstration, make sure that your terminal has been properly set up. See section 5.5.1, “Setting the Terminal Type”, for more information about setting up your terminal for use with vi.

5.2.1 Entering the Editor

To enter the editor and create a file named *temp*, type

```
vi temp
```

Your screen will look like this:

```
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
"temp" [New file]
```

Note that we show a twelve-line screen to save space. In reality, vi uses whatever size screen you have.

You are initially editing a copy of the file. The file itself is not altered until you save it. Saving a file is explained later in the demonstration. The top line of your display is the only line in the file and is marked by the cursor, shown above as an underline character. In this chapter, when the cursor is on a character that character will be enclosed in square brackets ([]).

The line containing the cursor is called the *current line*.

The lines containing tildes are not part of the file: they indicate lines on the screen only, not real lines in the file.

5.2.2 Inserting Text

To begin, create some text in the file *temp* by using the **i** (insert) command. To do this, press:

i

Next, type the following five lines to give yourself some text to experiment with. Press RETURN at the end of each line. If you make a mistake, use the BKSP key to erase the error and type the word again.

```
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.
```

```
~  
~  
~  
~  
~  
~
```

Press ESCAPE when you are finished.

Like most vi commands, the **i** command is not shown (or “echoed”) on your screen. The command itself switches you from command mode to insert mode.

When you are in *insert mode* every character you type is displayed on the screen. In *command mode* the characters you type are not placed in the file as text; they are interpreted as commands to be executed on the file. If you are not certain which mode you are in, press ESC until you hear the bell. When you hear the bell you are in command mode.

Once in insert mode, the characters you type are inserted into the file; they are *not* interpreted as vi commands. To exit insert mode and reenter command mode you will always press ESC. This switching between modes occurs often in vi, and it is important to get used to it now.

5.2.3 Repeating a Command

Next comes a command that you’ll use frequently in vi: the repeat command. The repeat command repeats the most recent insert or delete command. Since we have just executed an insert command, the repeat command repeats the insertion, duplicating the inserted text. The repeat command is executed by typing a period (.) or “dot”. So, to add five more lines of text, type “.”. The repeat command is repeated relative to the location of the cursor and inserts text *below* the current line. (Remember, the current line is always the line containing the cursor.) After you type dot (.), your screen will look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
```

~

5.2.4 Undoing a Command

Another command which is very useful (and which you’ll need often in the beginning) is the **undo** command, **u**. Press

u

and notice that the five lines you just finished inserting are deleted or “undone”.

Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.

~
~
~
~
~
~

Now type

u

again, and the five lines are reinserted! This undo feature can be very useful in recovering from inadvertent deletions or insertions.

5.2.5 Moving the Cursor

Now let's learn how to move the cursor around on the screen. In addition to the arrow keys, the following letter keys also control the cursor:

h Left

l Right

k Up

j Down

The letter keys are chosen because of their relative positions on the keyboard. Remember that the cursor movement keys only work in command mode.

Try moving the cursor using these keys. (First make sure you are in command mode by pressing the ESC key.) Then, type the **H** command to place the cursor in the upper left corner of the screen. Then type the **L** command to move to the lowest line on the screen. (Note that case is significant in our example: **L** moves to the lowest line on the screen; while **l** moves the cursor forward one character.) Next, try moving the cursor to the last line in the file with the goto command, **G**. If you type "2G", the cursor moves to the beginning of the second line in the file; if you have a 10,000 line file, and type "8888G", the cursor goes to the beginning of line 8888. (If you have a 600 line file and type "800G" the cursor doesn't move.)

These cursor movement commands should allow you to move around well enough for this demonstration. Other cursor movement commands you might want to try out are:

w	Moves forward a word
b	Backs up a word
0	Moves to the beginning of a line
\$	Moves to the end of a line

You can move through many lines quickly with the scrolling commands:

CNTRL-U Scrolls up 1/2 screen

CNTRL-D Scrolls down 1/2 screen

CNTRL-F Scrolls forward one screenful

CNTRL-B Scrolls backward one screenful

5.2.6 Deleting

Now that we know how to insert and create text, and how to move around within the file, we're ready to delete text. Many delete commands can be combined with cursor movement commands, as explained below. The most common delete commands are:

- dd Deletes the current line (the line the cursor is on), regardless of the location of the cursor in the line.
- dw Deletes the word above the cursor. If the cursor is in the middle of the word, deletes from the cursor to the end of the word.
- x Deletes the character above the cursor.
- d\$ Deletes from the cursor to the end of the line.
- D Deletes from the cursor to the end of the line.
- d0 Deletes from the cursor to the start of the line.
- . Repeats the last change. (Use this only if your last command was a deletion.)

To learn how all these commands work, we'll delete various parts of the demonstration file. To begin, press ESC to make sure you are in command mode, then move to the first line of the file by typing

1G

At first, your file should look like this:

```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

To delete the first line, type

dd

Your file should now look like this:

```
[T]ext contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~  
~  
~
```

Delete the word the cursor is sitting on by typing

dw

After deleting, your file should look like this:

```
[c]ontains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~  
~  
~
```

You can quickly delete the character above the cursor by pressing:

x

This leaves:

```
[o]ntains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~  
~  
~
```

Now type a “w” command to move your cursor to the beginning of the word “lines” on the first line. Then, to delete to the end of the line, type

d\$

Your file looks like this:


```

ontains_
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~

```

To delete all the characters on the line *before* the cursor type:

```
d0
```

This leaves a single space on the line:

```

_
Lines contain characters.
Files contain text.
Text contains lines.
Characters form words.
Words form text.
Lines contain characters.
Characters form words.
Words form text.
~
~
~

```

For review, let's restore the first two lines of the file.

Press "i" to enter insert mode, then type

```

Files contain text.
Text contains lines.

```

Press ESC to go back to command mode.

5.2.7 Searching for a Pattern

You can search forward for a pattern of characters by typing a slash (/) followed by the pattern you are searching for, terminated by a RETURN. For example, make sure you are in command mode (press ESC), then press

```
H
```

to move the cursor to the top of the screen. Now, type

```
/char
```

Don't press RETURN yet. Your screen should look like this:

```

Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
/char_

```

Press RETURN. The cursor moves to the beginning of the word “characters” on line three. To search for the next occurrence of the pattern “char”, press “n” (as in “next”). This will take you to the beginning of the word “characters” on the eighth line. If you keep pressing “n” vi searches past the end of the file, wraps around to the beginning, and again finds the “char” on line three.

Note that the slash character and the pattern that you are searching for appear at the bottom of the screen. This bottom line is the vi status line.

The *status line* appears at the bottom of the screen. It is used to display information, including patterns you are searching for, line-oriented commands (explained later in this demonstration), and error messages.

For example, to get status information about the file, press CNTRL-G. Your screen should look like this:

```

Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain [c]haracters.
Characters form words.
Words form text.
~
"temp" [Modified] line 4 of 10 --4%--

```

The status line on the bottom tells you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and your location in the file as a percentage of the number of lines in the file. The status line disappears as you continue working.

5.2.8 Searching and Replacing

Let's say you want to change all occurrences of “text” in the demonstration file to “documents”. Rather than search for “text”, then delete it and insert “documents”, you can do it all in one

command. The commands you have learned so far have all been *screen-oriented*. Commands that can perform more than one action (searching and replacing) are *line-oriented* commands.

Screen-oriented commands are executed at the location of the cursor. You do not need to tell the computer where to perform the operation; it takes place relative to the cursor. *Line-oriented* commands require you to specify an exact location (called an “address”) where the operation is to take place. Screen-oriented commands are easy to type in, and provide immediate feedback; the change is displayed on the screen. Line-oriented commands are more complicated to type in, but they can be executed independent of the cursor, and in more than one place in a file at a time.

All line-oriented commands are preceded by a colon which acts as a prompt on the status line. Line-oriented commands themselves are entered on this line and terminated with a RETURN.

In this chapter, all instructions for line-oriented commands will include the colon as part of the command.

To change “text” to “documents”, press ESC to make sure you are in command mode, then type:

```
:1,$s/text/documents/g
```

This command means “From the first line (1) to the end of the file (\$), find *text* and replace it with *documents* (s/text/documents/) everywhere it occurs on each line (g)”.

Press RETURN. Your screen should look like this:

```
Files contain documents.
Text contains lines.
Lines contain characters.
Characters form words.
Words form documents.
Files contain documents.
Text contains lines.
Lines contain characters.
Characters form words.
[W]ords form documents.
~
~
```

Note that “Text” in lines two and eight was not changed. Case is significant in searches.

Just for practice, use the undo command to change “documents” back to “text”. Press:

```
u
```

Your screen now looks like this:

```
[F]iles contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~
```

5.2.9 Leaving Vi

All of the editing you have been doing has affected a copy of the file, and *not* the file named *temp* that you specified when you invoked vi. To save the changes you have made, exit the editor and return to the XENIX shell, type

```
:x
```

Remember to press RETURN. The name of the file, and the number of lines and characters it contains are displayed on the status line:

```
"temp" [New file] 10 lines, 214 characters
```

Then the XENIX prompt appears.

5.2.10 Adding Text From Another File

In this section we'll create a new file, and insert text into it from another file. First, create a new file named *practice* by typing:

```
vi practice
```

This file is empty. Let's copy the text from *temp* and put it in *practice* with the line-oriented **read** command. Press ESC to make sure you are in command mode, then type

```
:r temp
```

Your file should look like this:

```
[F]iles contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~
```

The text from *temp* has been copied and put in the current file *practice*. There is an empty line at the top of the file. Move the cursor to the empty line and delete it with the **dd** command.

5.2.11 Leaving Vi Temporarily

Vi allows you to execute commands outside of the file you are editing, such as **date**. To find out the date and time, type

```
!:date
```

Press RETURN. This displays the date, then prompts you to press RETURN to reenter command mode. Go ahead and try it. Your screen should look similar to this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
!:date
Mon Jan 9 16:33:37 PST 1984
[Hit return to continue]_
```

5.2.12 Changing Your Display

Besides the set of editing commands described above, there are a number of options that can be set either when you invoke vi, or later when editing. These options allow you to control editing parameters such as line number display, and whether or not case is significant in searches. In this section we'll learn how to turn on line numbering, and how to look at the current option settings.

To turn on automatic line numbering, type

```
:set number
```

Press RETURN. Your screen is redrawn, and line numbers appear to the left of the text. Your screen looks like this:

```
1 Files contain text.
2 Text contains lines.
3 Lines contain characters.
4 Characters form words.
5 Words form text.
6 Files contain text.
7 Text contains lines.
8 Lines contain characters.
9 Characters form words.
10 Words form text.
~
~
```

You can get a complete list of the available options by typing

```
:set all
```

and pressing RETURN. Setting these options is described in section 5.5 “Setting Up Your Environment”, but it is important that you be aware of their existence. Depending on what you are working on, and your own preferences, you will want to alter the default settings for many of these options.

5.2.13 Canceling an Editing Session

Finally, to exit vi without saving the file *practice*, type:

`:q!`

and press RETURN. This cancels all the changes you have made to *practice* and, since it is a new file, deletes it. The prompt appears. If *practice* had already existed before this editing session, the changes you made would be disregarded, but the file would still exist.

This completes the demonstration. You have learned how to get in and out of vi, insert and delete text, move the cursor around, make searches and replacements, how to execute line-oriented commands, copy text from other files, and cancel an editing session. There are many more commands to learn, but the fundamentals of using vi have been covered. The following sections will give you more detailed information about these commands and about vi's other commands and features.

5.3 Editing Tasks

The following sections explain how to perform common editing tasks. By following the instructions in each section you will be able to complete each task described. Features that are needed in several tasks are described each time they are used, so some information is repeated.

5.3.1 How to Enter the Editor

There are several ways to begin editing, depending on what you are planning to do. This section describes how to start, or “invoke” the editor with one filename. To invoke vi on a series of files, see section 5.3.24, “Editing a Series of Files”.

5.3.1.1 With a Filename

The most common way to enter vi is to type “vi” and the name of the file you wish to edit:

`vi filename`

If *filename* does not already exist, a new, empty file is created.

5.3.1.2 At a Particular Line

You can also enter the editor at a particular place in a file. For example, if you wish to start editing a file at line 100, type:

`vi +100 filename`

The cursor is placed at line 100 of *filename*.

5.3.1.3 At a Particular Word

If you wish to begin editing at the first occurrence of a particular word, type

```
vi +/word filename
```

The cursor is placed at the first occurrence of *word*. For example, to begin editing the file *temp* at the the first occurrence of “contain”, type

```
vi +/contain temp
```

5.3.2 Moving the Cursor

The cursor movement keys allow you to move the cursor around in a file. Cursor movement can only be done in command mode.

5.3.2.1 Moving the Cursor By Characters: h,j,k,l,SPACE,BKSP

The SPACE bar and the l key move the cursor forward a specified number of characters. The BKSP key and the h key move it backward a specified number of characters. If no number is specified, the cursor moves one character. For example, to move backward four characters, type

```
4h
```

You can also move the cursor to a designated character on the current line. F moves the cursor back to the specified character, f moves it forward. The cursor rests on the specified character. For example, to move the cursor backward to the nearest *p* on the current line, type:

```
Fp
```

To move the cursor forward to the nearest *p*, type:

```
fp
```

The T and t keys work the same way as f and F, but place the cursor immediately before the specified character. For example, to move the cursor back to the space next to the nearest *p* in the current line, type

```
Tp
```

If the *p* were in the word *telephone*, the cursor would sit on the *h*.

The cursor always remains on the same line when you use these commands. If you specify a number greater than the number of characters on the line, the cursor does not move beyond the beginning or end of that line.

5.3.2.2 Moving the Cursor by Words: w, W, b, B, e, E

The w key moves the cursor forward to the beginning of the specified number of words. Punctuation and nonalphabetic characters (such as !@#\$%^&*()_+{}[]\`<>/) are considered words, so if a word is followed by a comma the cursor will count the comma in the specified number. For example, if the cursor rests on the first letter of the sentence

```
No, I didn't know he had returned.
```

and you press

```
6w
```

the cursor stops on the *k* in *know*.

W works the same way as w, but includes punctuation and nonalphabetic characters as part of the word. Using the above example, if you press

6W

the cursor stops on the *r* in *returned*; the comma and the apostrophe are included in their adjacent words.

The e and E keys move the cursor forward to the end of a specified number of words. The cursor is placed on the last letter of the word. The e command counts punctuation and nonalphabetic characters as separate words; E does not.

B and b move the cursor back to the beginning of a specified number of words. The cursor is placed on the first letter of the word. The b command counts punctuation and nonalphabetic characters as separate words; B does not. Using the above example, if the cursor is on the *r* in *returned*, type

4b

and the cursor moves to the *t* in *didn't*. Type

4B

and the cursor moves to the first *d* in *didn't*.

The w, W, b and B commands will move the cursor to the next line if that is where the designated word is, unless the current line ends in a space.

5.3.2.3 Moving the Cursor by Lines

Forward: j, CNTRL-N, +, RETURN, LINEFEED, \$

The RETURN, LINEFEED and + keys move the cursor forward a specified number of lines, placing the cursor on the first character. For example, to move the cursor forward six lines, type

6+

The j and CNTRL-N keys move the cursor forward a specified number of lines. The cursor remains in the same place on the line, unless there is no character in that place, in which case it moves to the last character on the line. For example, in the following two lines if the cursor is resting on the *e* in *characters*, pressing "j" moves it to the period at the end of the second line:

Lines contain characters.

Text contains lines.

The dollar sign(\$) moves the cursor to the end of a specified number of lines. For example, to move the cursor to the last character of the line four lines down from the current line, type:

4\$

Backward: k, CNTRL-P

CNTRL-P and k move the cursor backward a specified number of lines, keeping it on the same place on the line. For example, to move the cursor backward four lines from the current line, type

4k

5.3.2.4 Moving the Cursor on the Screen: H, M, L

The H, M and L keys move the cursor to the beginning of the top, middle and bottom lines of the screen, respectively.

5.3.3 Moving Around in a File: Scrolling

The following commands move the file so different parts can be displayed on the screen. The cursor is placed on the first letter of the last line scrolled.

5.3.3.1 Scrolling Up Part of the Screen: CNTRL-U

CNTRL-U scrolls up one-half screen.

5.3.3.2 Scrolling Up the Full Screen: CNTRL-B

CNTRL-B scrolls up a full screen.

5.3.3.3 Scrolling Down Part of the Screen: CNTRL-D

CNTRL-D scrolls down one-half screen.

5.3.3.4 Scrolling Down a Full Screen: CNTRL-F

CNTRL-F scrolls down a full screen.

5.3.3.5 Placing a Line at the Top of the Screen: z

To scroll the current line to the top of the screen, press

`z`

then press RETURN. To place a specific line at the top of the screen, precede the “z” with the line number, as in

`33z`

Press RETURN, and line 33 scrolls to the top of the screen. For information on how to display line numbers, see section 5.5.5, “Displaying Line Numbers: number”.

5.3.4 Inserting Text Before the Cursor: i and I

You can begin inserting text before the cursor anywhere on a line, or at the beginning of a line. In order to insert text into a file, you must be in “insert mode”. To enter insert mode press

`i`

The “i” does not appear on the screen. Any text typed after the “i” becomes part of the file you are editing. To leave insert mode and reenter command mode, press ESC. For more explanation of modes in vi, see section 5.2.2, “Inserting Text”.

5.3.4.1 Anywhere on a Line: i

To insert text before the cursor, use the **i** command. Press the **i** key to enter insert mode (the **i** does not appear on your screen), then begin typing your text. To leave insert mode and reenter command mode, press **ESC**.

5.3.4.2 At the Beginning of the Line: I

Using an uppercase **I** to enter insert mode also moves the cursor to the beginning of the current line. It is used to start an insertion at the beginning of the current line.

5.3.5 Appending After the Cursor: a and A

You can begin appending text after the cursor anywhere on a line, or at the end of a line. Press **ESC** to leave insert mode and reenter command mode.

5.3.5.1 Anywhere on a Line: a

To append text after the cursor, use the **a** command. Press the **a** key to enter insert mode (the “**a**” does not appear on your screen), then begin typing your text. Press **ESC** to leave insert mode and reenter command mode.

5.3.5.2 At the end of a Line: A

Using an uppercase **A** to enter insert mode also moves the cursor to the end of the current line. It is useful for appending text at the end of the current line.

5.3.6 Correcting Typing Mistakes

If you make a mistake while you are typing, the simplest way to correct it is with the **BKSP** key. Backspace across the line until you have backspaced over the mistake, then retype the line. You can only do this, however, if the cursor is on the same line as the error. See sections 5.3.12 through 5.3.15 for other ways to correct typing mistakes.

5.3.7 Opening a New Line

To open a new line above the cursor, press **O**. To open a new line below the cursor, press **o**. Both commands place you in insert mode, and you may begin typing immediately. Press **ESC** to leave insert mode and reenter command mode.

You may also use the **RETURN** key to open new lines above and below the cursor. To open a line above the cursor, move the cursor to the beginning of the line, press **i** to enter insert mode, then press **RETURN**. (For information on how to move the cursor, see section 5.3.2, “Moving the Cursor”.) To open a line below the cursor, move the cursor to the end of the current line, press **i** to enter insert mode, then press **RETURN**.

5.3.8 Repeating the Last Insertion

CNTRL-@ repeats the last insertion. Press “**i**” to enter insert mode, then press **CNTRL-@**.

CNTRL-@ only repeats insertions of 128 characters or less. If more than 128 characters were inserted, CNTRL-@ does nothing.

For other methods of repeating an insertion, see section 5.3.8, “Repeating the Last Insertion”, section 5.3.9, “Inserting Text From Other Files”, and section 5.3.22, “Repeating a Command”.

5.3.9 Inserting Text From Other Files

To insert the contents of another file into the file you are currently editing, use the **read** command. Move the cursor to the line immediately *above* the place you want the new material to appear, then type

```
:r filename
```

where *filename* is the file containing the material to be inserted, and press RETURN. The text of *filename* appears on the line below the cursor, and the cursor moves to the first character of the new text. This text is a copy; the original *filename* still exists.

Inserting selected lines from another file is more complicated. The selected lines are copied from the original file into a temporary holding place called a “buffer”, then inserted into the new file.

1. To select the lines to be copied, save your original file with the **write** command (:w), but do not exit vi.

2. Type

```
:e filename
```

where *filename* is the file that contains the text you want to copy, and press RETURN.

3. Move the cursor to the first line you wish to select.

4. Type

```
mk
```

This “marks” the first line of text to be copied into the new file with the letter “k”.

5. Move the cursor to the last line of the selected text. Type

```
"ay'k
```

The lines from your first “mark” to the cursor are placed, or “yanked” into buffer *a*. They will remain in buffer *a* until you replace them with other lines, or until you exit the editor.

6. Type

```
:e#
```

to return to your previous file. (For more information about this command, see section 5.3.25, “Editing a New File Without Leaving the Editor”.) Move the cursor to the line above the place you want the new text to appear, then type

```
"ap
```

This “puts” a copy of the yanked lines into the file, and the cursor is placed on the first letter of this new text. The buffer still contains the original yanked lines.

You can have 26 buffers named *a*, *b*, *c*, up to and including *z*. To name and select different buffers, replace the *a* in the above examples with whatever letter you wish.

You may also delete text into a buffer, then insert it in another place. For information on this type of deletion and insertion, see section 5.3.17, "Moving Text".

5.3.9.1 Copying Lines From Elsewhere in the File

To copy lines from one place in a file to another place in the same file, use the **co** (copy) command.

Co is a line-oriented command, and to use it you must know the line numbers of the text to be copied and its destination. To find out the number of the current line type

```
:nu
```

and press RETURN. The line number and the text of that line are displayed on the status line. To find out the destination line number, move the cursor to the line above where you want the copied text to appear and repeat the **:nu** command. You can also make line numbers appear throughout the file with the *linenumber* option. For information on how to set this option, see section 5.5.5, "Displaying Line Numbers: number". The following example uses the *linenumber* option.

```
1 [F]iles contain text.  
2 Text contains lines.  
3 Lines contain characters.  
4 Characters form words.  
5 Words form text.  
~  
~  
~  
~  
~  
~  
~
```

Using the above example, to copy lines 3 and 4 and put them between lines 1 and 2, type

```
:3,4 co 1
```

The result is:

```
1 Files contain text.  
2 Lines contain characters.  
3 [C]haracters form words.  
4 Text contains lines.  
5 Lines contain characters.  
6 Characters form words.  
7 Words form text.  
~  
~  
~  
~  
~
```

If you have text that is to be inserted several times in different places, you can save it in a temporary storage area, called a "buffer", and insert it whenever it is needed. For example, to repeat the first line of the following text after the last line:


```
[F]iles contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
```

```
~
~
~
~
~
~
~
```

1. Move the cursor over the *F* in *Files*. Type the following line, which will not be echoed on your screen:

```
"ayy
```

This “yanks” the first line into buffer *a*. Move the cursor over the *W* in *Words*.

2. Type the following line:

```
"ap
```

This “puts” a copy of the yanked line into the file, and the cursor is placed on the first letter of this new text. The buffer still contains the original yanked line.

Your screen looks like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
[F]iles contain text.
```

```
~
~
~
~
~
~
~
```

If you wish to “yank” several consecutive lines, indicate the number of lines you wish to yank after the name of the buffer. For example, to place three lines from the above text in the “a” buffer, type

```
"a3yy
```

For another method of placing text in a buffer, and more information about naming buffers, see section 5.3.9, “Inserting Text From Other Files”.

5.3.10 Inserting Control Characters into Text

Many control characters have special meaning in vi, even when typed in insert mode. To remove their special significance, press CNTRL-V before typing the control character. Note that CNTRL-J, CNTRL-Q, and CNTRL-S cannot be inserted as text. CNTRL-J is a newline character. CNTRL-Q and CNTRL-S are meaningful to the operating system, and are trapped by it before

they are interpreted by vi.

5.3.11 Joining and Breaking Lines

To join two lines press

J

while the cursor is on the first of the two lines you wish to join.

To break one line into two lines, position the cursor on the space preceding the first letter of what will be the second line, press

r

then press RETURN.

5.3.12 Deleting a Character: x and X

The **x** and **X** commands delete a specified number of characters. The **x** command deletes the character above the cursor; the **X** command deletes the character immediately before the cursor. If no number is given, one character is deleted. For example, to delete three characters following the cursor (including the character the above the cursor), type:

3x

To delete three characters preceding the cursor, type:

3X

5.3.13 Deleting a Word: dw

The **dw** command deletes a specified number of words. If no number is given, one word is deleted. A word is interpreted as numbers and letters separated by whitespace. When a word is deleted, the space after it is also deleted. For example, to delete three words, type:

3dw

5.3.14 Deleting a Line: D and dd

The **D** command deletes all text following the cursor on that line, including the character the cursor is resting on. The **dd** command deletes a specified number of lines and closes up the space. If no number is given, only the current line is deleted. For example, to delete three lines, type:

3dd

Another way to delete several lines is to use a line-oriented command. To use this command it helps to know the line numbers of the text you wish to delete. For information on how to display line numbers, see section 5.5.5, "Displaying Line Numbers: number".

For example, to delete lines 200 through 250, type

:200,250d

Press RETURN. When the command finishes, the message

50 lines

appears on the vi status line, indicating how many lines were deleted.

It is possible to remove lines without displaying line numbers using shorthand "addresses". For example, to remove all lines from the current line (the line the cursor rests on) to the end of the

file, type

`:$d`

The dot (.) represents the current line, and the dollar sign stands for the last line in the file. To delete the current line and 3 lines following it, type

`:$,+3d`

To delete the current line and 3 lines preceding it, type

`:$,-3d`

For more information on using addresses in line-oriented commands, see *vi(C)* in the *XENIX Reference Manual*.

5.3.15 Deleting an Entire Insertion

If you wish to delete all of the text you just typed, press CNTRL-U while you are in insert mode. The cursor returns to the beginning of the insertion. The text of the original insertion is still displayed, and any text you type replaces it. When you press ESC, any text remaining from the original insertion disappears.

5.3.16 Deleting and Replacing Text

Several vi commands combine removing characters and entering insert mode. The following sections explain how to use these commands.

5.3.16.1 Overstriking: r and R

The **r** command replaces the character under the cursor with the next character typed. To replace the character under the cursor with a “b”, for example, type:

`rb`

If a number is given before **r**, that number of characters is replaced with the next character typed. For example, to replace the character above the cursor, plus the next three characters, with the letter “b”, type

`4rb`

Note that you now have four “b”s in a row.

The **R** command replaces as many characters as you type, up to the end of the line. To end the replacement, press ESCAPE. For example, to replace the second line in the following text with “Spelling is important.”:

```
Files contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.
```

```
~  
~  
~  
~  
~  
~  
~  
~
```

Move the cursor over the *T* in *Text*. Press **R**, then type

Spelling is important.

Press **ESC** to end the replacement. If you make a mistake, use the **BKSP** key to correct it. Your screen should now look like this:

```
Files contain text.  
Spelling is important[.]  
Lines contain characters.  
Characters form words.  
Words form text.
```

```
~  
~  
~  
~  
~  
~  
~  
~
```

5.3.16.2 Substituting: s and S

The **s** command replaces a specified number of characters, beginning with the character under the cursor, with text you type in. For example, to substitute *xyz* for the cursor and two characters following it, type:

```
3sxyz
```

The **S** command deletes a specified number of lines and replaces them with text you type in. You may type in as many new lines of text as you wish; **S** affects only how many lines are deleted. If no number is given, one line is deleted. For example, to delete four lines, including the current line, type:

```
4S
```

This differs from the **R** command. The **S** command deletes the entire current line; the **R** command deletes text from the cursor onward.

5.3.16.3 Replacing a Word: cw

The **cw** command replaces a word with text you type in. For example, to replace the word *bear*

with the word *fox*, move the cursor over the *b* in *bear*. Press

`cw`

A dollar sign appears over the *r* in *bear*, marking the end of the text that is being replaced. Type

fox

and press ESC. The rest of *bear* disappears and only *fox* remains.

5.3.16.4 Replacing the Rest of a Line: C

The **C** command replaces text from the cursor to the end of the line. For example, to replace the text of the sentence

Who's afraid of the big bad wolf?

from *big* to the end, move the cursor over the *b* in *big* and press

C

A dollar sign (\$) replaces the question mark (?) at the end of the line. Type the following:

little lamb?

Press ESC. The remaining text from the original sentence disappears.

5.3.16.5 Replacing a Whole Line: cc

The **cc** command deletes a specified number of lines, regardless of the location of the cursor, and replaces them with text you type in. If no number is given, the current line is deleted.

5.3.16.6 Replacing a Particular Word on a Line

If a word occurs several times on one line, it is often convenient to use a line-oriented command to replace it. For example, to replace the word *removing* with *deleting* in the following sentence:

In vi, removing a line is as easy as removing a letter.

Make sure the cursor is at the beginning of that line, and type

`:s/removing/deleting/g`

Press RETURN. This line-oriented command means "Substitute (s) for the word *removing* the word *deleting*, everywhere it occurs on the current line (g)". If you don't include a g at the end, only the first occurrence of *removing* is changed.

For more information on using line-oriented commands to replace text, see section 5.3.19, "Searching and Replacing".

5.3.17 Moving Text

To move a block of text from one place in a file to another, you can use the line-oriented **m** command. You must know the line numbers of your file to use this command. The *linenumber* option displays line numbers. To set this option, press ESC to make sure you are in command mode, then type:

`set linenumber`

XENIX User's Guide

Line numbers will appear to the left of your text. (For more information on setting the *linenumber* option, see section 5.5.5, "Displaying Line Numbers: number".)

The following example uses the *linenumber* option. For other ways to display line numbers, see section 5.3.29, "Finding Out What Line You're On".

```
1 [F]iles contain text.  
2 Text contains lines.  
3 Lines contain characters.  
4 Characters form words.  
5 Words form text.  
~  
~  
~  
~  
~  
~  
~
```

To insert lines 2 and 3 between lines 4 and 5, type

```
:2,3m4
```

Your screen should look like this:

```
1 Files contain text.  
2 Characters form words.  
3 Text contains lines.  
4 Lines contain characters.  
5 [W]ords form text.  
~  
~  
~  
~  
~  
~  
~
```

To place line 5 after line 2, type

```
:5m2
```

After moving, your screen should look like this:


```

1 Files contain text.
2 Characters form words.
3 [W]ords form text.
4 Text contains lines.
5 Lines contain characters.
~
~
~
~
~
~
~

```

To make line 4 the first line in the file, type

```
:4m0
```

Your screen should look like this:

```

1 [T]ext contains lines.
2 Files contain text.
3 Characters form words.
4 Words form text.
5 Lines contain characters.
~
~
~
~
~
~
~

```

You can also delete text into a temporary storage place, called a “buffer”, and insert it wherever you wish. When text is deleted it is placed in a “delete buffer”. There are nine “delete buffers”.

The first buffer always contains the most recent deletion. In other words, the first deletion in a given editing session goes into buffer 1. The second deletion also goes into buffer 1, and pushes the contents of the old buffer 1 into buffer 2. The third deletion goes into buffer 1, pushing the contents of buffer 2 into buffer 3, and the contents of buffer 1 into buffer 2. When buffer 9 has been used, the next deletion pushes the current text of buffer 9 off the stack and it disappears.

Text remains in the delete buffers until it is pushed off the stack, or until you quit the editor, so it is possible to delete text from one file, change files without leaving the editor, and place the deleted text in another file.

Delete buffers are particularly useful when you wish to remove text, store it, and put it somewhere else. Using the following text as an example

```
[F]iles contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~  
~  
~  
~  
~  
~  
~
```

Delete the first line by typing

`dd`

Delete the third line the same way. Now move the cursor to the last line in the example and press

`"1p`

The line from the *second* deletion appears:

```
Text contains lines.  
Characters form words.  
Words form text.  
[L]ines contain characters.  
~  
~  
~  
~  
~  
~  
~  
~
```

Now type:

`"2p`

The line from the *first* deletion appears:

```
Text contains lines.  
Characters form words.  
Words form text.  
Lines contain characters.  
[F]iles contain text.  
~  
~  
~  
~  
~  
~  
~
```

Inserting text from a delete buffer does not remove the text from the buffer. Since the text remains in a buffer until it is either pushed off the stack or until you quit the editor, you may use it as many times as you wish.

It is also possible to place text in named buffers. For information on how to create named buffers, see section 5.3.9, “Inserting Text From Other Files”.

5.3.18 Searching: / and ?

You can search forward and backward for patterns in vi. To search forward, press the slash (/) key. The slash appears on the status line. Type the characters you wish to search for. Press RETURN. If the specified pattern exists, the cursor will move to the first character of the pattern. For example, to search forward in the file for the word *account*, type:

```
/account
```

Press RETURN. The cursor is placed on the first character of the pattern. To place the cursor at the beginning of the line above *account*, for example, type

```
/account/-
```

To place the cursor at the beginning of the line two lines above the line that contains *account*, type

```
/account/-2
```

To place the cursor two lines below *account*, type

```
/account/+2
```

To search backward through a file, use ? instead of / to start the search. For example, to find all occurrences of *account* above the cursor, type:

```
?account
```

To search for a pattern containing any of the special characters (. * \ [] ~ \$ and ^), each special character must be preceded by a backslash. For example, to find the pattern *U.S.A.*, type:

```
/U\.S\.A\./
```

You can continue to search for a pattern by pressing

```
n
```

after each search. The pattern is unaffected by intervening vi commands, and you can use **n** to search for the pattern until you type in a new pattern or quit the editor.

Vi searches for exactly what you type. If the pattern you are searching for contains an uppercase letter (for example, if it appears at the beginning of a sentence), vi ignores it. To disregard case in a search command, you can set the ignorecase option:

```
:set ignorecase
```

By default, searches “wrap around” the file. That is, if a search starts in the middle of a file, when vi reaches the end of the file it will “wrap around” to the beginning, and continue until it returns to where the search began. Searches will be completed faster if you specify forward or backward searches, depending on where you think the pattern is.

If you do not want searches to wrap around the file, you can change the “wrapscan” option setting. Type:

```
:set nowrapscan
```

and press RETURN to prevent searches from wrapping. For more information about setting options, see section 5.5, “Setting Up Your Environment”.

5.3.19 Searching and Replacing

The search and replace commands allow you to perform complex changes to a file in a single command. Learning how to use these commands is a must for the serious user of vi.

The syntax of a search and replace command is:

```
g/pattern1/s/[pattern2]/[options]
```

Brackets indicate optional parts of the command line. The *g* tells the computer to execute the replacement on every line in the file. Otherwise the replacement would occur only on the current line. The *options* are explained in the following sections.

To explain these commands we will use the example file from the demonstration run:

```
[F]iles contain text.  
Text contains lines.  
Lines contain characters.  
Characters form words.  
Words form text.  
~  
~  
~  
~  
~  
~  
~
```

5.3.19.1 Replacing a Word

To replace the word *contain* with the word *are* throughout the file, type the following command:

```
:g/contain /s//are /g
```

This command says "On each line of the file (*g*), find *contain* and substitute for that word (*s//*) the word *are*, everywhere it occurs on that line (the second *g*)". Note that a space is included in the search pattern for *contain*; without the space *contains* would also be replaced.

After the command executes your screen should look like this:

```
[F]iles are text.  
Text contains lines.  
Lines are characters.  
Characters form words.  
Words form text.  
~  
~  
~  
~  
~  
~  
~
```

5.3.19.2 Printing all Replacements

To replace *contain* with *are* throughout the file, then print every line changed, use the *p* option:

```
:g/contain /s//are /gp
```

Press RETURN. After the command executes, each line in which *contain* was replaced by *are* is

printed on the lower part of the screen. To remove these lines, redraw the screen by pressing CNTRL-L

5.3.19.3 Choosing a Replacement

Sometimes you may not want to replace every instance of a given pattern. The **c** option displays every occurrence of *pattern* and waits for you to confirm that you want to make the substitution. If you press *y* the substitution takes place; if you press RETURN the next instance of *pattern* is displayed.

To run this command on the example file, type

```
:g/contain/s//are/gc
```

Press RETURN. The first instance of *contain* appears on the status line:

```
Files containtext.
```

Press *y*, then RETURN. The next occurrence of *contain* appears.

5.3.20 Pattern Matching

Search commands often require, in addition to the characters you want to find, a context in which you want to find them. For example, you may want to locate every occurrence of a word at the beginning of a line. Vi provides several special characters that specify particular contexts.

5.3.20.1 Matching the Beginning of a Line

When a caret (^) is placed at the beginning of a pattern, only patterns found at the beginning of a line are matched. For example, the following search pattern only finds *text* when it occurs as the first word on a line:

```
/^text/
```

To search for a caret that appears as text you must precede it with a backslash (\).

5.3.20.2 Matching the End of a Line

When a dollar sign (\$) is placed at the end of a pattern, only patterns found at the end of a line are matched. For example, the following search pattern only finds *text* when it occurs as the last word on a line:

```
/text$/
```

To search for a dollar sign that appears as text you must precede it with a backslash (\).

5.3.20.3 Matching Any Single Character

When used in a search pattern, the period (.) matches any single character except the newline character. For example, to find all words that end with *ed*, use the following pattern:

```
/.ed /
```

Note the space between the *d* and the backslash.

To search for a period in the text, you must precede it with a backslash (\).

5.3.20.4 Matching a Range of Characters

A set of characters enclosed in square brackets matches any single character in the range designated. For example, the search pattern

```
/[a-z]/
```

finds any lowercase letter. The search pattern

```
/[aA]pple/
```

finds all occurrences of *apple* and *Apple*.

To search for a bracket that appears as text, you must precede it with a backslash (\).

5.3.20.5 Matching Exceptions

A caret (^) at the beginning of *string* matches every character *except* those specified in *string*. For example the search pattern

```
[^a-z]
```

finds anything but a lowercase letter or a newline.

5.3.20.6 Matching the Special Characters

To place a caret, hyphen or square bracket in a search pattern, precede it with a backslash. To search for a caret, for example, type:

```
/\^/
```

If you need to search for many patterns that contain special characters, you can reset the *magic* option. To do this, type

```
:nomagic
```

This removes the special meaning from the characters `.`, `\`, `$`, `[` and `]`. You can include them in search and replace commands without a preceding backslash. Note that the special meaning cannot be removed from the special characters star (*) and caret (^); these must always be preceded by a backslash in searches.

To restore *magic*, type

```
:set magic
```

For more information about setting options, see section 5.5, "Setting Up Your Environment".

5.3.21 Undoing a Command: *u*

Any editing command can be reversed with the "undo" command. **Undo** works on both screen-oriented and line-oriented commands. For example, if you have deleted a line and then decide you wish to keep it, press *u* and the line will reappear. Use the following line as an example:


```
[T]ext contains lines.  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

Place the cursor over the *c* in *contains*, then delete the word with the **dw** command. Your screen should look like this:

```
Text [l]ines.  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

Press *u* to undo the **dw** command. *Contains* reappears:

```
Text [c]ontains lines.  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

If you press *u* again, *contains* is deleted again:

```
Text [l]ines.
```

```
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

It is important to remember that **u** only undoes the *last* command. For example, if you make a global search and replace, then delete a few characters with the **x** command, pressing **u** will undo the deletions but not the global search and replace.

5.3.22 Repeating a Command: .

Any screen-oriented vi command can be repeated with the **repeat** (.) command. For example, if you have deleted two words by typing:

```
2dw
```

you may repeat this command as many times as you wish by pressing the period key (.). Cursor movement does not affect the repeat command, so you may repeat a command as many times and in as many places in a file as you wish.

The repeat command only repeats the last vi command. Careful planning can save time and effort. For example, if you want to replace a word that occurs several times in a file (and for some reason you do not wish to use a global command), use the **cw** command instead of deleting the word with the **dw** command, then inserting new text with the **i** command. By using the **cw** command you can repeat the replacement with the dot (.) command. If you delete the word, then insert new text, dot only repeats the replacement.

5.3.23 Leaving the Editor

There are several ways to exit the editor and save any changes you may have made to the file. One way is to type:

```
:x
```

and press RETURN. This command replaces the old copy of the file with the new one you have just edited, quits the editor, and returns you to the XENIX shell. Similarly, if you type

```
ZZ
```

the same thing happens, except the old copy file is written out *only* if you have made any changes. Note that the **ZZ** command is *not* preceded by a colon, and is not echoed on the screen.

To leave the editor without saving any changes you have made to the file, type

```
:q!
```

The exclamation point tells vi to quit unconditionally. If you leave out the exclamation point:

```
:q
```

vi will not let you quit. You will see the error message:

No write since last change (:quit! overrides)

This message tells you to use “:q!” if you really want to leave the editor without saving your file.

5.3.23.1 Saving a File Without Leaving the Editor

There are many occasions when you must save a file without leaving the editor, such as when starting a new shell, or moving to another file. Before you can perform these tasks you must first save the current file with the **write** command:

```
:w
```

You do not need to type the name of the file; vi remembers the name you used when you invoked the editor. If you invoked vi without a filename, you may name the file by typing

```
:w filename
```

where *filename* is the name of the new file.

5.3.24 Editing a Series of Files

Entering and leaving vi for each new file takes time, particularly on a heavily used system, or when you are editing large files. If you have many files to edit in one session, you can invoke vi with more than one filename, and thus edit more than one file without leaving the editor, as in:

```
vi file1 file2 file3 file4 file5 file6
```

But typing out many filenames is tedious, and you may make a mistake. If you mistype a filename, you must either backspace over to mistake and retype the line, or kill the whole line and retype it. It is more convenient to invoke vi using the special characters as abbreviations.

To invoke vi on the above files without typing each name, type:

```
vi file*
```

This invokes vi on all files that begin with the letters *file*. You can plan your filenames to save time in later editing. For example, if you are writing a document that consists of many files, it would be wise to give each file the same filename extension, such as “.s”. Then you can invoke vi on the entire document:

```
vi *.s
```

You can also invoke vi on a selected range of files:

```
vi [3-5]*.s
```

or

```
vi [a-h]*
```

To invoke vi on all files that are five letters long, and have any extension:

```
vi ?????.*
```

For more information on using special characters, see Chapter 3 of this manual, section 3.3.4, “Special Characters”.

When you invoke vi with more than one filename, you will see the following message when the first file is displayed on the screen:

```
x files to edit
```

After you have finished editing a file, save it with the **write** command (:w), then go to the next file with the **next** command:

:n

The next file appears, ready to edit. It is not necessary to specify a filename; the files are invoked in alphabetical (or numerical, if the filenames begin with numbers) order.

If you forget what files you are editing, type:

:args

The list of files appears on the status line. The current file is enclosed in square brackets.

To edit a file out of order, such as *file4* after *file2*, type

:e file4

instead of using the **next** command. If you type:

:n

after you finish editing *file4*, you will go back to *file3*.

If you wish to start again from the beginning of the list, type:

:rew

To discard the changes you made and start again at the beginning, type:

:rew!

5.3.25 Editing a New File Without Leaving the Editor

You can start editing another file anywhere on the XENIX system without leaving vi. This saves time when you wish to edit several files in one session that are in different directories, or even in the same directory. For example, if you have finished editing */usr/joe/memo* and you wish to edit */usr/mary/letter*, first save the file *memo* with the **w** command (:w), then type:

:e /usr/mary/letter

/usr/mary/letter appears on your screen just as though you had left vi.

Note

You *must* write out your file with the **write** command (:w) if you want to save the changes you have made. If you try to edit a second file without writing out the first file, the message “No write since last change (:e! overrides)” appears. If you use :e! all your changes to the first file are discarded.

If you want to switch back and forth between two files, vi remembers the name of the last file edited. Using the above example, if you wish to go back and edit the file */usr/joe/memo* after you have finished with */usr/mary/letter*, type

:e#

The cursor is positioned in the same location it was when you first saved */usr/joe/memo*.

5.3.26 Leaving the Editor Temporarily: Shell Escapes

You can execute any XENIX command from within vi using the shell “escape” (as in, “escape from vi”) command, !. For example, if you wish to find out the date and time, type:


```
!:date
```

The exclamation point sends the remainder of the line to the shell to be executed, and the date and time appear on the vi status line. You can use the ! to perform any XENIX command. To send mail to joe without leaving the editor, type

```
!:mail joe
```

Type your message and send it. (For more information about the XENIX mail system, see Chapter 6, “Mail”.) After you send it, the message

```
[Hit return to continue]
```

appears. Press RETURN to continue editing.

If you want to perform several XENIX commands before returning to the editor, you can invoke a new shell:

```
!:sh
```

The XENIX prompt appears. You may execute as many commands as you like. Press CNTRL-D to terminate the new shell and return to your file.

If you have not written out your file before a shell escape, you will see the message:

```
[No write since last change]
```

It is a good idea to save your file with the **w**rite command (:w) before executing an escape, just in case something goes wrong. However, once you become an experienced vi user, you may wish to turn off this message. To turn off the “No write” message, reset the *warn* option, as follows:

```
:set nowarn
```

For more information about setting options in vi, see section 5.5, “Setting Up Your Environment”.

5.3.27 Performing a Series of Line-Oriented Commands: Q

If you have several line-oriented commands to perform, you can place yourself temporarily in line-oriented mode by typing

```
Q
```

while you are in command mode. A colon prompt appears on the status line.

Commands executed in this mode cannot be undone with the **u** command, nor do they appear on the screen until you re-enter normal vi mode. To re-enter normal vi mode, type

```
vi
```

5.3.28 Finding Out What File You’re In

If you forget what file you are editing, press CNTRL-G while you are in command mode. A line similar to the following appears on the status line:

```
“memo” [Modified] line 12 of 100 --12%--
```

From left to right, the following information is displayed:

- The name of the file
- Whether or not the file has been modified

- The line number the cursor is on
- How many lines there are in the file
- Your location in the file (expressed as a percentage)

This command is also useful when you need to know the line number of the current line for a line-oriented command.

The same information can be obtained by typing

`:file`

or

`:f`

5.3.29 Finding Out What Line You're On

To find out what line of the file you are on, type

`:nu`

and press RETURN. This command displays the current line number and the text of the line.

To display line numbers for the entire file, see section 5.5.5, "Displaying Line Numbers: number"

5.4 Solving Common Problems

The following is a list of common problems that you may encounter when using vi, along with the probable solution.

- **I don't know which mode I'm in.**

Press ESC until the bell rings. When the bell rings you are in command mode.

- **I can't get out of a subshell.**

Press CNTRL-D to exit any subshell. If you have created more than one subshell (not a good idea, usually), keep pressing CNTRL-D until you see the message:

[Hit return to continue]

- **I made an inadvertent deletion (or insertion).**

Press "u" to undo the last delete or insert command.

- **There are extra characters on my screen.**

Press CNTRL-L to redraw the screen.

- **When I type, nothing happens.**

Vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, slowly type:

`stty sane`

then press CNTRL-J or LINEFEED. Pressing CNTRL-J instead of RETURN is important here, since it is quite possible that the RETURN key will not work as a

newline character. To make sure that other terminal characteristics have not been altered, log off, turn your terminal off, turn your terminal back on, and then log back in. This should guarantee that your terminal's characteristics are back to normal. This procedure may vary somewhat depending on the terminal.

— **The system crashed while I was editing.**

Normally, vi will inform you (by sending you mail) that your file has been saved before a crash. The file can be recovered by typing

```
vi -r filename
```

If vi was unable to save the file before the crash, it is irretrievably lost.

— **I keep getting a colon on the status line when I press RETURN**

You are in line-oriented command mode. Type

```
vi
```

to return to normal vi command mode.

— **I get the error message “Unknown terminal type [Using open mode]” when I invoke vi.**

Your terminal type is not set correctly. To leave open mode, press ESC, then type

```
:wq
```

and press RETURN. Turn to section 5.5.1, “Setting the Terminal Type” for information on how to set your terminal type correctly.

5.5 Setting Up Your Environment

There are a number of options that can be set that affect your terminal type, how files and error messages are displayed on your screen, and how searches are performed. These options can be set with the **set** command while you are editing, or they can be placed in the vi startup file, *.exrc*. (The *.exrc* file is explained in section 5.5.13.) The following sections describe the most commonly used options and how to set them. There is a complete list of options in *vi(C)* in the *XENIX Reference Manual*.

5.5.1 Setting the Terminal Type

Before you can use vi, you must set the terminal type, if this has not already been done for you, by defining the **TERM** variable in your *.profile* file. (The *.profile* file is explained in the *XENIX User's Guide*.) The **TERM** variable is a number that tells the operating system what type of terminal you are using. To determine this number you must find out what type of terminal you are using. Then look up this type in *Terminals(M)* in the *XENIX Reference Manual*. If you cannot find your terminal type or its number, consult your System Administrator.

For these examples, we will suppose that you are using an HP 2621 terminal. For the HP 2621, the **TERM** variable is “2621”. How you define this variable depends on which shell you are using. You can usually determine which shell you are using by examining the prompt character. The Bourne shell prompts with a dollar sign (\$); the C-shell prompts with a percent sign (%).

5.5.1.1 Setting the TERM variable: The Visual Shell

If you are using the Visual Shell the terminal type has already been set, and you do not need to change it.

5.5.1.2 Setting the TERM variable: The Bourne Shell

To set your terminal type to 2621 place the following commands in the file *.profile*:

```
TERM=2621
export TERM
```

5.5.1.3 Setting the TERM variable: The C Shell

To set your terminal type to 2621 for the C shell, place the following command in the file *.login*:

```
setenv TERM 2621
```

5.5.2 Setting Options: The set Command

The **set** command is used to display option settings and to set options.

5.5.2.1 Listing the Available Options

To get a list of the options available to you and how they are set, type

```
:set all
```

Your display should look similar to this:

noautoindent	open	noslowopen
autoprint	nooptimize	tabstop=8
noautowrite	paragraphs=IPLPPPQPP LIbp	taglength=0
nobeautify	noprompt	ttytype=h19
directory=/tmp	noreadonly	term=h19
noerrorbells	redraw	noterse
hardtabs=8	report=5	warn
noignorecase	scroll=4	window=8
nolisp	sections=NHSHH HU	wrapscan
nolist	shell=/bin/sh	wrapmargin=0
magic	shiftwidth=8	nowriteany
nonumber	noshowmatch	

This chapter discusses only the most commonly used options. For information about the options not covered in this chapter, see *vi(C)* in the *XENIX Reference Manual*.

5.5.2.2 Setting an Option

To set an option, use the **set** command. For example, to set the *ignorecase* option so that case is *not* ignored in searches, type

```
:set noignorecase
```


5.5.3 Displaying Tabs and End-of-Line: *list*

List causes the “hidden” characters and end-of-line to be displayed. The default setting is *nolist*. To display these characters, type

```
:set list
```

Your screen is redrawn. The dollar sign (\$) represents end-of-line and control-I (^I) represents the tab character.

5.5.4 Ignoring Case in Search Commands: *ignorecase*

By default, case is significant in search commands. To disregard case in searches, type

```
:set ignorecase
```

To change this option, type

```
:set noignorecase
```

5.5.5 Displaying Line Numbers: *number*

It is often useful to know the line numbers of a file. To display these numbers, type:

```
:set number
```

This redraws your screen. Numbers appear to the left of the text. To remove line numbers, type:

```
:set nonumber
```

5.5.6 Printing the Number of Lines Changed: *report*

The *report* option tells you the number of lines modified by a line-oriented command. For example,

```
:set report=1
```

reports the number of lines modified, if more than one line is changed. The default setting is

```
report=5
```

which reports the number of lines changed when more than five lines are modified.

5.5.7 Changing the Terminal Type: *term*

If you are logged in on a terminal that is a different type than the one you normally use, you can check the terminal type setting by typing:

```
:set term
```

Press RETURN. See section 5.5.1, “Setting the Terminal Type” for more information about TERM variables.

5.5.8 Shortening Error Messages: *terse*

After you become experienced with vi, you may want to shorten your error messages. To change from the default (*noterse*), type

```
:set terse
```

As an example of the effect of *terse*, when *terse* is set the message

No write since last change, quit! overrides

becomes

No write

5.5.9 Turning Off Warnings: *warn*

After you become experienced with vi, you may want to turn off the error message that appears if you have not written out your file before a shell escape (:!) command. To turn these messages off, type

```
:set nowarn
```

5.5.10 Permitting Special Characters in Searches: *nomagic*

The *nomagic* option allows the inclusion of the special characters (. \ \$ []) in search patterns without a preceding backslash. This option does *not* affect caret (^) or star (*); they must be preceded by a backslash in searches regardless of *magic*. To set *nomagic*, type

```
:set nomagic
```

5.5.11 Limiting Searches: *wrapscan*

By default, searches in vi “wrap” around the file until they return to the place they started. To save time you may want to disable this feature. Use the following command:

```
:set nowrapscan
```

When this option is set, forward searches go only to the end of the file, and backward searches stop at the beginning.

5.5.12 Turning on Messages: *mesg*

If someone sends you a message with the **write** command while you are in vi the text of the message will appear on your screen. To remove the message from your display you must press CNTRL-L. When you invoke vi, write permission to your screen is automatically turned off, preventing **write** messages from appearing. If you wish to receive **write** messages while in vi, reset this option as follows:

```
:set mesg
```

5.5.13 Customizing Your Environment: The *.exrc* File

Each time vi is invoked, it reads commands from the file named *.exrc* in your home directory. This file sets your preferred options so that they do not need to be set each time you invoke vi. A sample *.exrc* file follows:

```
set number
set ignorecase
set nowarn
set report=1
```

Each time you invoke vi with the above options, your file is displayed with line numbers, case is ignored in searches, warnings before shell escape commands are turned off, and any command

that modifies more than one line will display a message indicating how many lines were changed.

5.6 Summary of Commands

The following tables contain all the basic commands discussed in this chapter.

Entering Vi

Typing this:	Does this:
<code>vi file</code>	Starts at line 1
<code>vi +n file</code>	Starts at line <i>n</i>
<code>vi + file</code>	Starts last line
<code>vi +/pattern file</code>	Starts at <i>pattern</i>
<code>vi -r file</code>	Recovers <i>file</i> after a system crash

Cursor Movement

Pressing this key:	Does this:
<code>h</code>	Moves 1 space left
<code>l</code>	Moves 1 space right
<code>SPACEBAR</code>	Moves 1 space right
<code>w</code>	Moves 1 word right
<code>b</code>	Moves 1 word left
<code>k</code>	Moves 1 line up
<code>j</code>	Moves 1 line down
<code>RETURN</code>	Moves 1 line down
<code>)</code>	Moves to end of sentence
<code>(</code>	Moves to beginning of sentence
<code>}</code>	Moves to beginning of paragraph
<code>{</code>	Moves to end of paragraph
<code>CNTRL-W</code>	Moves to first character of insertion
<code>CNTRL-U</code>	Scrolls up 1/2 screen
<code>CNTRL-D</code>	Scrolls down 1/2 screen
<code>CNTRL-F</code>	Scrolls down one screen
<code>CNTRL-B</code>	Scrolls up one screen

Inserting Text

Pressing	Starts insertion:
i	Before the cursor
I	Before first character on the line
a	After the cursor
A	After last character on the line
o	On next line down
O	On the line above
r	On current character, replaces one character only
R	On current character, replaces until ESC

Delete Commands

Command	Function
dw	Deletes a word
d0	Deletes to beginning of line
d\$	Deletes to end of line
3dw	Deletes 3 words
dd	Deletes the current line
5dd	Deletes 5 lines

Change Commands

Command	Function
cw	Changes 1 word
3cw	Changes 3 words
cc	Changes current line
5cc	Changes 5 lines

Search Commands

Command	Function	Example
/and	Finds the next occurrence of <i>and</i>	and, stand, grand
?and	Finds the previous occurrence of <i>and</i>	and, stand, grand
/^The	Finds next line that starts with <i>The</i>	The, Then, There
/[bB]ox/	Finds the next occurrence of <i>box</i> or <i>Box</i>	
n	Repeats the most recent search, in the same direction	

Search and Replace Commands

Command	Result	Example
:s/pear/peach/g	All <i>pears</i> become <i>peach</i> on the current line	
:1,\$s/file/directory	Replaces <i>file</i> with <i>directory</i> from line 1 to the end.	<i>filename</i> becomes <i>directoryname</i>
:g/one/s//1/g	Replaces every occurrence of <i>one</i> with 1.	one becomes 1, oneself becomes 1self, someone becomes some1

Pattern Matching: Special Characters

This character:	Matches:
^	Beginning of a line
\$	End of a line
.	Any single character
[]	A range of characters

Leaving Vi

Command	Result
:w	Writes out the file
:x	Writes out the file, quits vi
:q!	Quits vi without saving changes
!:command	Executes <i>command</i>
!:sh	Forks a new shell
!!command	Executes <i>command</i> and places output on current line
:e <i>file</i>	Edits <i>file</i> (save current file with :w first)

Options

This option:	Does this:
all	Lists all options
term	Sets terminal type
ignorecase	Ignores case in searches
list	Displays tab and end-of-line characters
number	Displays line numbers
report	Prints number of lines changed by a line-oriented command
terse	Shortens error messages
warn	Turns off “no write” warning before escape
nomagic	Allows inclusion of special characters in search patterns without a preceding backslash
nowrapscan	Prevents searches from wrapping around the end or beginning of a file.
mesg	Permits display of messages sent to your terminal with the write command

Chapter 6

Mail

6.1	Introduction	1
6.2	Demonstration	2
6.2.1	Composing and Sending a Message	2
6.2.2	Reading Mail	2
6.2.3	Leaving Mail	3
6.3	Basic Concepts	3
6.3.1	Mailboxes	4
6.3.2	Messages	4
6.3.3	Modes	4
6.3.4	Message-Lists	5
6.3.5	Headers	6
6.3.6	Command Syntax	6
6.4	Using Mail	7
6.4.1	Entering and Exiting Mail	7
6.4.2	Sending Mail	7
6.4.3	Reading Mail	8
6.4.4	Disposing of Mail	8
6.4.5	Composing Mail	8
6.4.6	Forwarding Mail	9
6.4.7	Replying to Mail	9
6.4.8	Specifying Messages	9
6.4.9	Creating Mailing Lists	9
6.4.10	Sending Network Mail	10
6.4.11	Setting Options	10
6.5	Commands	10
6.5.1	Getting Help: help and ?	10
6.5.2	Reading Mail: p, +, -, and <i>restart</i>	11
6.5.3	Finding Out the Number of the Current Message: =	12
6.5.4	Displaying the First Five Lines: t	12
6.5.5	Displaying Headers: h	12
6.5.6	Deleting Messages: d and dp	12
6.5.7	Undeleting Messages: u	13
6.5.8	Leaving mail: q and x	13
6.5.9	Saving Your Mail: s	13
6.5.10	Saving Your Mail: w	13
6.5.11	Saving Your Mail: mb	14
6.5.12	Saving Your Mail: ho	14
6.5.13	Printing Your Mail on the Lineprinter: l	14
6.5.14	Sending Mail: m	14
6.5.15	Replying to Mail: r and R	14
6.5.16	Forwarding Mail: f and F	14
6.5.17	Creating Mailing Lists: a	15
6.5.18	Setting and Unsetting Options: se and uns	15
6.5.19	Editing a message: e and v	15

6.5.20	Executing Shell Commands: sh and !	15
6.5.21	Finding Out the Number of Characters in a Message: si	16
6.5.22	Changing the Working Directory: cd	16
6.5.23	Reading Commands From a File: so	16
6.6	Leaving Compose Mode Temporarily	16
6.6.1	Getting Help: ~?	16
6.6.2	Printing the Message: ~p	17
6.6.3	Editing the Message: ~e and ~v	17
6.6.4	Editing Headers: ~t, ~c, ~b, ~s, ~R and ~h	17
6.6.5	Adding a File to the Message: ~r and ~d	18
6.6.6	Enclosing Another Message: ~m and ~M	18
6.6.7	Saving the Message in a File: ~w	18
6.6.8	Leaving Mail Temporarily: ~! and ~	18
6.6.9	Escaping to Mail Command Mode: ~:	19
6.6.10	Placing a Tilde at the Beginning of a Line: ~~	19
6.7	Setting Up Your Environment: The .mailrc File	20
6.7.1	The Subject Prompt: asksubject	20
6.7.2	The CC Prompt: askcc	20
6.7.3	Printing the Next Message: autoprint	20
6.7.4	Listing Messages in Chronological Order: chron and mchron	20
6.7.5	Using the Period to Send a Message: dot	21
6.7.6	Sending Mail While in Mail: execmail	21
6.7.7	Including Yourself in a Group: metoo	21
6.7.8	Saving Aborted Messages: save	21
6.7.9	Printing the Version Header: quiet	21
6.7.10	Choosing an Editor: The EDITOR String	21
6.7.11	Choosing an Editor: The VISUAL String	21
6.7.12	Choosing a Shell: The SHELL String	22
6.7.13	Changing the Escape Character: The escape String	22
6.7.14	Setting Page Size: The page String	22
6.7.15	Saving Outgoing Mail: The record String	22
6.7.16	Keeping Mail in the System Mailbox: autombox	22
6.7.17	Changing the top Value: The toplines String	22
6.7.18	Sending Mail Over Telephone Lines: ignore	22
6.8	Using Advanced Features	23
6.8.1	Command Line Options	23
6.8.2	Using Mail as a Reminder Service	23
6.8.3	Handling Large Amounts of Mail	24
6.8.4	Maintenance and Administration	24
6.9	Quick Reference	25
6.9.1	Command Summary	25
6.9.2	Compose Escape Summary	28
6.9.3	Option Summary	29

6.1 Introduction

The XENIX **mail** system is a versatile communication facility that allows XENIX users to compose, send, receive, forward, and reply to mail. Users can also create distribution groups and send copies of messages to multiple users. These functions are integrated into XENIX so that all users can quickly and easily communicate with each other.

This chapter is organized to satisfy the needs of both the beginning and advanced user. The first sections discuss basic concepts, tasks, and commands. Later sections discuss advanced topics and provide quick reference to the **mail** program's many functions. The major sections in this chapter are:

Demonstration	Shows new users how to get started.
Basic Concepts	Discusses the fundamental ideas and terminology used in mail .
Using Mail	Shows how to perform common mailing procedures such as composing, sending, forwarding, and replying to mail.
Commands	Discusses each mail command.
Leaving Compose Mode Temporarily	Discusses and gives examples of each command available when composing a message. These commands are called compose escapes.
Setting Up Your Environment	Discusses the user's mail startup file and options that may be set to customize functions.
Using Advanced Features	Discusses advanced features such as using mail as a reminder service and handling a large volume of mail.
Quick Reference	Summarizes all commands, compose escapes, and options.

6.2 Demonstration

The **mail** command lets you perform two distinct functions: sending mail and disposing of mail. In this demonstration, we will show you how to send mail to yourself, read a message, delete, it, and exit the **mail** program.

6.2.1 Composing and Sending a Message

To begin, type

mail self

where "self" is your user name. Next, type the following lines, each terminated with a RETURN:

This is a message sent to myself.

I compose a message by entering lines of text.

The message is ended by typing CNTRL-D on a newline.

As you enter the message you can use *compose escapes* to perform special functions. To get a list of the available compose escapes, type

~?

on a new line. To specify a subject, use the ~s escape. For example, type:

~s Sample subject

To specify a list of people to receive carbon copies use the ~c escape. For example, type

~c abel

To view the message as it will appear when you send it, type:

~p

This will print the following:

Message contains:

To: self

Subject: Sample subject

Cc: abel

This is a message sent to myself.

I compose a message by entering lines of text.

The message is ended by typing CNTRL-D on a newline.

Finally, to end the message and send it to those you have mentioned in the *To:* and the *Cc:* fields, press CNTRL-D by itself on a line. This will exit the **mail** program and return you to the XENIX shell. Once you have sent mail, there is no way to undo the act, so be careful.

6.2.2 Reading Mail

Within a short time, you should receive the message:

You have mail.

(You must press RETURN before this message will appear on your screen.) This message informs you that the message you have just sent has arrived in your system mailbox. To read this message and any others that may have been sent to you, type

mail

Mail then displays a sign-on message and a list of message headers that look something like this:

Mail version 3.0 August 30, 1982. Type ? for help.

1 message:

1 self Fri Aug 31 12:26 7/188 "Sample subject"

—
When there is more than one message in your mailbox, the *most recent* message is displayed at the top of the list. Messages are numbered in ascending order from least to most recent, so the message at the top of the list (the most recent message) has the highest number. The message header includes who sent the message, when it was sent, the number of lines and characters, and the subject of the message. The underscore prompt prompts you to enter a **mail** command. Now type

?

to get help on all the available **mail** commands. Next, type

p

to see the message that you sent to yourself. **Mail** prints the following:

From self Fri Aug 20 12:26:52 1982

To: self

Subject: Sample subject

This is a message sent to myself.

I compose a message by entering lines of text.

The message is ended by typing CNTRL-D on a newline.

Note that the message you sent to yourself now contains information about the sender of the message-- a line telling who sent the message and when it was sent. The next line tells who the message was sent to. A subject and carbon copy (Cc:) field can be specified by the sender. If they are present, they too are displayed when you read the message.

6.2.3 Leaving Mail

Since this message has no real use, you can delete it by typing:

d

To get out of **mail**, type:

q

Mail then displays the message

0 messages held in /usr/spool/mail/self

and returns you to the XENIX shell.

This ends the demonstration. For more detailed information, see the discussions in following sections.

6.3 Basic Concepts

It is much easier to use **mail** if you understand the basic concepts that underlie it. The concepts discussed in this section are:

— Mailboxes

— Messages

- Modes
- Command syntax

6.3.1 Mailboxes

It is useful to think of the **mail** system as modeled after a typical postal system. What is normally called a post office is called the *system mailbox* in this chapter. The system mailbox contains a file for each user in the directory */usr/spool/mail*. Your own personal or *user mailbox* is the file named *mbox* in your home directory. Mail sent to you is put in your system mailbox; you may choose to save mail in your user mailbox after you have read it. Note that the user mailbox differs from a *real* mailbox in several respects:

1. *You* decide whether mail is to be placed in the user mailbox; it is not automatically placed there.
2. The user mailbox is *not* the place where mail is initially routed—that place is the system mailbox in the directory */usr/spool/mail*.
3. Mail is not picked up *from* your user mailbox.

6.3.2 Messages

In **mail**, the message is the basic unit of exchange between users. Messages consist of two parts: a heading and a body. The heading contains the following fields:

- To:** This field is mandatory and contains one or more valid user names corresponding to real users to whom you may send mail.
- Subject:** This optional field contains text describing the message.
- Cc:** The carbon copy field contains one or more valid names of those who are to receive copies of a message. Message recipients see these names in the received message. This field can be empty.
- Bcc:** The blind carbon copy field contains the one or more valid names of people who are to receive copies of a message. Recipients do *not* see these names in the received messages. This field can be empty.

Return-receipt-to:

The return receipt to: field contains the valid name or names of those who are to receive an automatic acknowledgement of the message. This field can be empty.

The body of a message is text exclusive of the heading. The body can be empty.

6.3.3 Modes

Often, the biggest hurdle to using **mail** is understanding what modes of operation are available. This section discusses each mode.

When you invoke **mail** you are using the shell. If you want to mail a letter without entering **mail** command mode, you can do so by typing

```
mail john <letter
```

Here, the file *letter* is sent to the user *john*.

Note

Be very careful when mailing a file with the input redirection symbol (<). If you accidentally type the output redirection symbol (>), you will overwrite the file, destroying its contents.

You can enter a message from your shell by typing:

```
mail john
```

Next, enter the text of your message as follows:

This is the text of the message.

Press RETURN to start a new line, then CNTRL-D to send the message. Messages such as the one above are created in **mail**'s *compose mode*. When entering text in compose mode, there are several special keys associated with line editing functions: these are the same special characters that are available to you when executing normal XENIX commands. For example, you can kill the line you are editing by typing the kill character, normally a CNTRL-U. To backspace, press either CNTRL-H or the BACKSPACE key. From compose mode, you can issue commands called compose escapes. These are also called *tilde escapes* because the command letters are preceded by a tilde (~). When you execute these commands you are temporarily leaving or escaping from compose mode; hence the name. Note that once you've pressed RETURN to end a line, you cannot change that line from within compose mode; to change it, you must enter edit mode.

The most common way of using **mail** is to just type

```
mail
```

This automatically places you in **mail command mode**. In this mode, you are prompted by an underscore for commands that permit you to manipulate your mail.

You can enter *edit mode* from either compose mode or command mode. In edit mode, you edit the body of a message using the full capabilities of an editor. To enter edit mode from command mode, use either the **e** or **edit** command to enter *ed*, or the **v** or **visual** command to enter *vi*. (*Vi* may not be available on your system.) To enter edit mode from compose mode, use the compose escapes *~e* and *~v*, respectively.

6.3.4 Message-Lists

Many **mail** commands take a list of messages as an argument. A *message-list* is a list of message numbers, ranges, and names, separated by spaces or tabs. Message numbers may be either decimal numbers, which directly specify messages, or one of the special characters *^*, *.*, or *\$*, which specify the first, current, or last *undeleted* message, respectively. Here, relevant means *not deleted*.

A range of messages is two message numbers separated by a dash. To print the first four messages on the screen, type

```
p 1-4
```

and to print all the messages from the current message to the last message, type

```
p .-$
```

A *name* is a user name. Messages can be printed by specifying the name of the sender. For example, to print each message sent to you by *john*, type

```
p john
```


As a shorthand notation, you can specify star (*) to get all *undeleted* messages. Thus,

p *

prints all messages except those that have been deleted,

d *

deletes all messages, and

u *

undeletes all deleted messages. (All three of these commands are described later in detail in Section 6.5 "Commands.")

6.3.5 Headers

When you enter **mail**, a list of message *headers* is displayed. A header is a single line of text containing descriptive information about a message. (Note that we use the word *heading* to describe the first part of a message, and *header* to describe **mail**'s one-line description of a message.) The information includes:

- The number of the message
- The sender
- The date sent
- The number of characters and lines
- The subject (if the message contains a *Subject:* field)

Message headers are displayed in *windows* with the **headers** command. A header window contains no more than 18 headers. If there are fewer than 18 messages in the mailbox, all are displayed in one header window. If there are *more* than 18 messages, then the list is divided into an appropriate number of windows. You can move forward and backward one window at a time with the

headers +

and

headers -

commands.

6.3.6 Command Syntax

Each **mail** command has its own syntax. Some take no arguments, some take only one, and others take several arguments. The more flexible commands, such as **print**, accept combinations of message-lists and user names. For these commands, **mail** first gathers all message numbers and ranges, then finds all messages from any specified user names. The full message-list is the intersection of these two sets of messages. Thus, the message-list "4-15 miller" matches all messages between 4 and 15 that are from miller.

Each **mail** command is typed on a line by itself, and any arguments follow the command word. The command need not be typed in its entirety—the first command that matches the typed prefix is used. For example, you can type "p" instead of "print" for the **print** command and "h" instead of "headers" for the **headers** command.

After the command itself is typed, one or more spaces should be entered to separate the command from its arguments. If a **mail** command does not take arguments, any arguments you give are ignored and no error occurs. For commands that take message-lists as arguments, if no message-list is given, the last message printed is used. If it does not satisfy the requirements of the command, the search proceeds forward. If there are no messages forward of the current message, the search proceeds backwards, and if there are no good messages at all, **mail** types:

No applicable messages

6.4 Using Mail

This section describes how to perform some basic tasks when using **mail**. More detailed discussions of each of these commands are presented in later sections.

6.4.1 Entering and Exiting Mail

To begin a session with **mail**, type:

mail

The headers for each received message are then displayed one screenful at a time. To display the next screenful of headers (if any), type

h+

To end the **mail** session, use the **quit** (**q**) command. All messages remain in the system mailbox unless they have been deleted with the **delete** (**d**) command, saved with the **save** or **write** command, or held in your user mailbox with the **mbox** command. Deleted messages are discarded. The **-f** command line option causes **mail** to read in the contents of *mbox*. Optionally, a filename may be given as an argument to **-f**, so that the specified file is read, instead. When you **quit**, **mail** writes all messages back to this file.

If you send mail over a noisy phone line, you will notice that many of the bad characters turn out to be the RUBOUT or DEL character, which causes **mail** to abort messages. To deal with this annoyance, you can invoke **mail** with the **-i** option which causes these bad characters to be ignored.

6.4.2 Sending Mail

To send a message, invoke **mail** with the names of the people and groups you want to receive the message. Next, type in your message. When you are finished, press CNTRL-D at the beginning of a line. The message is automatically sent to the specified people. While entering the text of your message, you can escape to an editor or perform other useful functions with compose escapes. Section 6.4.5, "Composing Mail," describes some features of **mail** available to help you when composing messages.

If you have a file that contains a written message, you can send it to sam, bob, and john by typing:

mail sam bob john <letter

where *letter* is the name of the file you are sending.

Note

Be very careful when mailing a file with the input redirection symbol (<). If you accidentally type the output redirection symbol (>), you will overwrite the file, destroying its contents.

If **mail** cannot be delivered to a specified address, you will either be notified immediately, in which case a copy of the undeliverable message is appended to the file *dead.letter*, or you will be notified via return mail, in which case a copy is included in the return mail message.

6.4.3 Reading Mail

To read messages sent to you, type

`mail`

Mail then checks your mail out of the system mailbox and prints out a one-line header of each message, one screenful at a time (to view the next screenful, type "h+"). The most recent message is initially the first message (numbered highest, because messages are numbered chronologically) and may be printed using the **print** command. You can move forward one message by pressing RETURN or typing '+'. To move forward *n* messages use "+*n*". You can move backwards one message with the "-" command or move backwards *n* messages and print with "-*n*". You can also move to any arbitrary message and print it by typing its number.

If new messages arrive while you are in **mail**, the following message appears:

New mail has arrived--type 'restart' to read.

Type

`restart`

and the headers of the new messages are displayed.

6.4.4 Disposing of Mail

After examining a message you can delete it with the **delete** (**d**) command, reply to it with the **reply** (**r**) command, forward it with the **forward** (**f**) command, or skip to the next message by pressing RETURN. Deletion causes the mail program to forget about the message. This is not irreversible; the message can be *undeleted* with the **undelete** (**u**) command by typing

`u number`

6.4.5 Composing Mail

To compose mail, you must enter compose mode. Do this from XENIX command level by typing

`mail john`

where john is the name of a user to whom you want to send mail. From **mail** command mode, you can enter compose mode with the **mail**, **reply**, or **Reply** commands. Once in compose mode, the text that you type is appended one line at a time to the body of the message you are sending. Normal line editing functions are available when entering text, including CNTRL-U to kill a line and BACKSPACE to back up one character. Note that entering two interrupts in a row (i.e., pressing INTERRUPT twice), aborts your composition.

While you are composing a message, **mail** treats lines beginning with the tilde (~) in a special way. This character introduces commands called compose escapes. For example, typing

```
~m
```

by itself on a line places a copy of the most recently printed message inside the message you are composing. The copy is shifted right one tabstop. Other escapes set up heading fields, add and delete recipients to the message, allow you to escape to an editor, let you revise the message body, or run XENIX commands. To get a list of the available compose escapes when in compose mode, type:

```
~?
```

See also Section 6.6, “Leaving Compose Mode Temporarily,” later in this chapter.

6.4.6 Forwarding Mail

To forward a message, use the **forward (f)** command. For example, type

```
f john
```

to place a copy of the current message inside a new message. The copy is shifted right one tabstop, and the new message is forwarded to John. John will receive a message heading indicating that you have forwarded the message. The **Forward (F)** command works just like its lowercase counterpart, except that the forwarded message is not shifted right one tabstop.

6.4.7 Replying to Mail

You can use the **reply** command to set up a response to a message, automatically addressing a reply to the person who sent the original message. You then type in text and send the message by pressing CTNRL-D on a line by itself. The **Reply** command works just like its lowercase counterpart, except that the message is sent to others named in the original message’s *To:* and *Cc:* fields.

6.4.8 Specifying Messages

Commands such as **print** and **delete** can be given a message-list argument to apply to several messages at once. Thus “delete 2 3” deletes messages 2 and 3, while “delete 1–5” deletes messages 1 through 5. A star (*) addresses all messages, and a dollar sign (\$) addresses the last (highest numbered) message. The **top (t)** command prints the first five lines of a message; hence, you can type

```
top *
```

to print the first five lines of every message. Message-lists can contain combinations of lists, ranges, and names. For example, the following command prints out all messages from tom or bob and numbered 2, 4, 10, 11, or 12:

```
p tom bob 2 4 10-12
```

6.4.9 Creating Mailing Lists

You can create personal mailing lists so that, for example, you can send mail to *cohorts* and have it go to a group of people. Such lists are defined by placing an *alias* line like

```
alias cohorts bill bob barry
```

in the file *.mailrc* in your home directory. The current list of such aliases can be displayed with the **alias (a) mail** command. Personal aliases are expanded in mail sent to others so that they will

be able to **Reply** to each individual recipient. For example, the *To:* field in a message sent to *cohorts* will read

To: bill bob barry

and not

To: cohorts

Normally, system-wide aliases are available to all users. These are installed by whoever is in charge of your system. For more information, see section 6.8, "Using Advanced Features," later in this chapter.

6.4.10 Sending Network Mail

Mail can be sent between XENIX machines connected with Micnet by specifying a machine name and the user name on that machine, separated by a colon:

machine:user

If appropriate gateways are known to your system, you can send mail to sites within the UUCP network using the syntax:

machine!user

(Be sure to escape the ! by preceding it with a backslash (\) when giving it on a *cs*h command line.) Mail may also interpret other characters in the **mail** path when dealing with other networks. In most cases, aliases should be set up so that specifying machine names is unnecessary. For more information about sending network mail, see the *XENIX Operations Guide*. For more information about UUCP, see the *XENIX Programmer's Reference*.

6.4.11 Setting Options

Mail has several options that you can **set** from **mail** command mode or in the file *.mailrc* in your home directory. For example, "set askcc" enables the **askcc** switch and causes prompting for additions to the *Cc:* field when you finish composing a message. These and other options are discussed in Section 6.7 "Setting Up Your Environment: The *.mailrc* File."

6.5 Commands

This section describes each of the commands available to you in **mail** command mode. The examples in this section assume you have invoked **mail** and that you have several messages you want to dispose of. Note that in general, **mail** commands can be invoked with either the name of the command or a one- or two-character mnemonic abbreviation. In the text of the command descriptions below, this mnemonic abbreviation is enclosed in parentheses after the name of the command. All commands are printed in boldface, except in the examples.

6.5.1 Getting Help: **help** and **?**

The **help** (?) command prints out a brief summary of all **mail** commands, so if you ever get stuck when you are in **mail** command mode, type:

?

or

help

6.5.2 Reading Mail: p, +, -, and restart

To look at a specific message, use the **print (p)** command. For example, pretend you have a header-list that looks like this:

```
3 john   Wed Sep 21 09:21 26/782 "Notice"
2 sam    Tue Sep 20 22:55 6/83  "Meeting"
1 tom    Mon Sep 19 01:23 6/84  "Invite"
```

Reading from the left, each header contains the message number, who sent it, the day, date, and time it was sent, the number of lines and characters in the message, and its subject.

To examine the second message, type:

```
p 2
```

This might cause **mail** to respond with:

```
Message 2:
From sam  Tue Sep 20 22:55 1983
Subject: Meeting
```

```
Meeting everyone, please don't forget!
```

To look at message 3, type

```
-
```

or to look at message 1, type

```
+
```

The commands **+** and **-** execute relative to the last message referred to, which in our example was 2. For large numbers of messages, you can skip forward and backward by the number of messages specified as an argument to **+** and **-**. For example, typing

```
+3
```

skips forward three messages. If you type

```
p *
```

then all messages are displayed, since the star (*) matches all messages.

Pressing RETURN prints out the next message in the header-list. You can always go to a message and print it by giving its message number or one of the special characters, caret (^), dot (.), or dollar sign (\$). In the example where message 2 is the current message

```
.
```

prints the current message,

```
^
```

prints message 1, and

```
$
```

prints message 3.

When new mail arrives while you are in **mail**, the message "New mail has arrived—type 'restart' to read." If you wish to read the new messages, type

```
restart
```

The headers of the new messages appear.

6.5.3 Finding Out the Number of the Current Message: =

The **number (=)** command prints out the message number of the current message. It takes no arguments.

6.5.4 Displaying the First Five Lines : t

The **top (t)** command takes a message-list and prints the first five lines of each addressed message. For example

```
top 2-12
```

prints out the first five lines of each of the messages 2 through 12. Note that the number of lines printed out by **top** can be set with the *toplines* option.

6.5.5 Displaying Headers: h

The **headers (h)** command displays header windows or lists of headers. A header window contains no more than 18 headers. With no argument, the **headers** command displays a header window in which the current message header is displayed at the center of the window.

To examine the next set of 18 headers, type:

```
h +
```

To examine the previous set, type:

```
h -
```

Both plus and minus take an optional numeric argument that indicates the number of header windows to move forward or backward before printing. If a message-list is given, then the **headers** command prints out the header line for each message in the list, disregarding all windowing. For example

```
h joe
```

displays all the message headers from joe. The following are some characteristics of the header-list:

- Deleted messages do not appear in the listing.
- Messages saved with the **save** command are flagged with a star (*).
- Messages to be saved in your user mailbox are flagged with an "M".
- If the *autombox* option is set, messages held with the **hold** command are flagged with an "H".

6.5.6 Deleting Messages: d and dp

Unless you indicate otherwise, each message you receive is automatically saved in the system mailbox when you quit **mail**. Often, however, you don't want to save messages you have received. To delete messages, use the **delete (d)** command. For example,

```
delete 1
```

prevents **mail** from retaining message 1 in the system mailbox. The message will disappear altogether, along with its number.

The **dp** command deletes the current message and prints the next message. It is useful for quickly reading and disposing of mail. Using **dp** is the same as using the **d** command with the *autoprint* option set. See also the **undelete** command, below.

6.5.7 Undeleting Messages: u

The **undelete (u)** command causes a message that has been previously deleted with **d** or **dp** to reappear as if it had never been deleted. For example, to undelete message 3, type

```
u3
```

You cannot undelete messages from previous **mail** sessions; they are gone for good.

6.5.8 Leaving mail : q and x

When you have read all your messages, you can leave **mail** with the **quit (q)** command. All messages are held in your system mailbox, except the following:

- Deleted messages, which are discarded irretrievably.
- Messages marked with the **mbox** command, which are saved in *mbox* in your home directory (i.e., your user mailbox).
- Messages saved with the **save** and **write** commands are deleted from the system mailbox. Forwarded messages are *not* deleted.

Note that if the *autombox* option is set, messages that you have read are automatically saved in your user mailbox. If you wish to leave **mail** quickly without altering either your system or user mailbox, you can use the **exit (x)** command. This returns you to the shell without changing anything: no messages are deleted or saved. Files that you invoke with the **mail -f** switch are unaffected as well.

6.5.9 Saving Your Mail: s

The **save (s)** command lets you save messages to files other than *mbox*. By using **save**, you can organize your mail by putting messages in appropriate files. The **save** command writes out each message to the file given as the last argument on the command line. For example, the following command appends messages 1-5 to the file *letters*:

```
s 1-5 letters
```

The file *letters* is created if it does not already exist. Saved messages are not automatically retained in the system mailbox when you quit, nor are they selected by the **print** command described above, unless explicitly requested. Each saved message is marked with a star (*).

Save writes out the entire message, including the *To:*, *Subject:*, and *Cc:* fields. In comparison, the **write** command, discussed below, writes out only the bodies of the specified messages.

6.5.10 Saving Your Mail: w

The **write (w)** command writes out *the body* of each message to the file given as the last argument on the command line. Each written message is marked with a star (*). The syntax is similar to that of the **save** command. For example,

```
w 3-17 john elliot book
```

writes out the bodies of all messages from john and elliot in the number range 3-17. They are

concatenated to the end of the file named *book*.

6.5.11 Saving Your Mail: **mb**

The **mbox (mb)** command marks each message specified in a message-list, so that all are saved in the user mailbox when a **quit** command is executed. Message headers are marked with an "M" to show that they are to be saved in *mbox*.

6.5.12 Saving Your Mail: **ho**

The **hold (ho)** command takes a message-list and marks each message so that it is saved in your system mailbox instead of deleted or saved in *mbox* when you quit. Saving of files in the system mailbox happens by default, so use **hold only** when you have also set the *autombox* option.

6.5.13 Printing Your Mail on the Lineprinter: **l**

The **lpr (l)** command paginates and prints out messages to the lineprinter. It takes a message-list as its argument, then paginates and prints out each message. For example

```
l doug
```

prints out each message from the user doug on the lineprinter.

6.5.14 Sending Mail: **m**

To send mail to a user, use the **mail (m)** command. This sends mail in the manner described for the **reply** command, except that you supply a list of recipients either as an argument or by entering them in the *To:* field. All compose escapes work in **mail**. Note that the **mail** command is in most ways identical to typing *mail users* at the XENIX command level.

6.5.15 Replying to Mail: **r** and **R**

Often, you want to deal with a message by responding to its author right away. The **reply (r)** command is useful for this purpose: it takes a message-list and sends mail to the author of each message. The original message's subject field is copied as the reply's subject. Each message is composed in compose mode; thus all compose escapes work in **reply**, and messages are terminated by pressing CNTRL-D.

The **Reply (R)** command works just like its lowercase counterpart, except that copies of the reply are also sent to everyone shown in the original message's *To:* and *Cc:* fields.

6.5.16 Forwarding Mail: **f** and **F**

To forward a copy of a message, use the **forward (f)** command. This causes a copy of the current message to be sent to the specified users. The message is marked as saved, and then deleted from the system mailbox when you exit mail. For example, to forward the current message to someone whose login name is john, type

```
f john
```

John will receive the forwarded message, along with a heading showing that you are the one who forwarded it. Inside the new message, the forwarded message is indented one tab stop. An optional message number can also be given. For example,

f 2 john bill

forwards message 2 to john and bill.

The **Forward (F)** command is identical to the lowercase **forward** command, except that the forwarded message is not indented.

6.5.17 Creating Mailing Lists: a

The **alias (a)** command links a group of names with the single name given by the first argument, thus creating a mailing list. For example, you could type

```
alias beatles john paul george ringo
```

so that whenever you used the name *beatles* in a destination address (as in “mail beatles”), it would be expanded so that you are really referring to the four names aliased to *beatles*. With no arguments, **alias** prints out all currently-defined aliases. With one argument, it prints out the users defined by the given alias.

You will probably want to define aliases in the startup file, *.mailrc*, so that you don’t have to redefine them each time you invoke **mail**. See section 6.7, “Setting Up Your Environment: The *.mailrc* File”, for more information.

6.5.18 Setting and Unsetting Options: se and uns

Mail switch and string options can be set with the **mail** commands **set** and **unset**. A switch option is either on or off (set or unset). String options are strings of characters that are assigned values with the syntax *option=string*. Multiple options may be specified on a line. It is most useful to place **set** and **unset** commands in the file *.mailrc* in your home directory, where they become your own personal default options when you invoke **mail**. For example, you might have a **set** command that looked like this:

```
set dot metoo topline=10 SHELL=/usr/bin/sh
```

The options *dot* and *metoo* are switch options; *toplines* and *SHELL* are string options.

The command

```
set ?
```

prints out a list of the available options. See the section “Setting Up Your Environment,” for descriptions of these options.

6.5.19 Editing a message: e and v

To edit individual messages using the text editor, use the **edit (e)** command. It takes a message-list and processes each message in turn by writing it to a temporary file. The editor, *ed*, is then automatically invoked so that you can edit the temporary file. When you finish editing the message, write the message out, then quit the editor. **Mail** reads the message back into the message buffer and removes the temporary file.

It is often useful to be able to invoke either a line or visual editor, depending on the type of terminal you are using. To invoke **vi**, you can use the **visual (v)** command. The operation of the **visual** command is otherwise identical to that of the **edit** command.

6.5.20 Executing Shell Commands: sh and !

To execute a shell command without leaving **mail**, precede the command with an exclamation point. For example

!date

prints out the current date without leaving **mail**. To enter a new shell, type:

sh

To exit from this new shell and return to **mail** command mode, press CNTRL-D.

6.5.21 Finding Out the Number of Characters in a Message: si

The **size (si)** command prints out the number of characters in each message in a message-list. For example, the command

si 1-4

might print out:

4: 234
3: 1000
2: 23
1: 456

6.5.22 Changing the Working Directory: cd

The **cd** command changes the working directory to the name of the directory you give it as an argument. If no argument is given, the directory is changed to your home directory. This command works just like the normal XENIX **cd** command. (Note that exiting **mail** returns you to the directory from which you entered **mail**; thus the **mail cd** command works only within **mail**.) You may want to place a **cd** command in your **.mailrc** file so that you always begin executing **mail** from within the same directory.

6.5.23 Reading Commands From a File: so

The **source (so)** command reads in **mail** commands from the file. Normally, these commands are **alias**, **set**, and **unset** commands.

6.6 Leaving Compose Mode Temporarily

While composing a message to be sent to others, it is often useful to print a message, invoke the text editor on a partial message, execute a shell command, or perform some other function. **Mail** provides these capabilities through *compose escapes* (sometimes called *tilde escapes*) which consist of a tilde (~) at the beginning of a line, followed by a single character that specifies the function to be performed. These escapes are available *only* when you are composing a new message. They have no meaning when you are in **mail** command mode. The available compose escapes are described below.

6.6.1 Getting Help: ~?

The help escape is the first compose escape you should know because it tells you about all the others. For example, if you type

~?

a brief summary of the available compose escapes is printed on your screen. Note that **~h** prompts for heading fields and does *not* give help.

6.6.2 Printing the Message: ~p

To print the current text of a message you are composing, type:

~p

This prints a line of dashes and the heading and body of the message so far.

6.6.3 Editing the Message: ~e and ~v

If you are dissatisfied with a message as it stands, you can edit the message by invoking the editor, **ed**, with the editor escape, ~e. This causes the message to be copied into a temporary file so that you can edit it. Similarly, the ~v escape causes the message to be copied into a temporary file so that you can edit it with the **vi** editor. After modifying the message to your satisfaction, write it out and quit the editor. **Mail** responds by typing

(continue)

after which you may continue composing your message.

6.6.4 Editing Headers: ~t, ~c, ~b, ~s, ~R and ~h

To add additional names to the list of message recipients, type the escape:

~t name1 name2 ...

You can name as many additional recipients as you wish. Note that users originally on the recipient list will still receive the message: you cannot remove anyone from the recipient list with ~t. To remove a recipient, use the ~h command, which is discussed later in this section.

You can replace or add a subject field by using the ~s escape:

~s *line-of-text*

This replaces any previous subject with *line-of-text*. The subject, if given, appears near the top of the message, prefixed with the heading *Subject:*. You can see what the message looks like by using ~p, which prints out all heading fields along with the body of the text.

You may occasionally prefer to list certain people as recipients of carbon copies of a message rather than direct recipients. The escape

~c name1 name2 ...

adds the named people to the *Cc:* list. The escape

~cc name1 name2 ...

performs an identical function. Similarly, the escape

~b name1 name2 ...

adds the named people to the *Bcc:* (Blind carbon copy) list. The people on this list receive a copy of the message, but are not mentioned anywhere in the message you send. Remember that you can always execute a ~p escape to see what the message looks like.

The escape

~R

adds or changes the person or persons named in the return-receipt to: field.

The recipients of the message are given in the *To:* field; the subject is given in the *Subject:* field, carbon copy recipients are given in the *Cc:* field and the return receipt recipient in the *Return-receipt-to:* field. If you wish to edit these in ways impossible with the ~t, ~s, ~c, and ~R escapes,

you can use

`~h`

where `h` stands for “heading.” The escape `~h` prints *To:* followed by the current list of recipients and leaves the cursor at the end of the line. If you type in ordinary characters, they are appended to the end of the current list of recipients. You can also use the normal XENIX command line editing characters to edit these fields, so you can erase existing heading text by backspacing over it.

When you press RETURN, **mail** advances to the *Subject:* field, where the same rules apply. Another RETURN brings you to the *Cc:* field, another brings you to the *Bcc:* field, and yet another to the *Return-receipt-to:* field. Each of these fields can be edited in the same way. Finally, another RETURN leaves you appending text to the end of your message body. As always, you can use `~p` to print the current text of the heading fields along with the body of the message.

6.6.5 Adding a File to the Message: `~r` and `~d`

It is often useful to be able to include the contents of some file in your message. The escape

`~r filename`

is provided for this purpose, and causes the named file to be appended to your current message. **Mail** complains if the file doesn't exist or can't be read. If the read is successful, **mail** prints the number of lines and characters appended to your message.

As a special case of `~r`, the escape

`~d`

reads in the file *dead.letter* in your home directory. This is often useful because **mail** copies the text of your message buffer to *dead.letter* whenever you abort the creation of a message by either typing two consecutive interrupts or entering a `~q` escape.

6.6.6 Enclosing Another Message: `~m` and `~M`

If you are sending mail from within mail's command mode, you can insert a message sent to you into the message you are currently composing. For example, you might type:

`~m 4`

This reads message 4 into the message you are composing, shifted right one tab stop. The escape

`~M 4`

performs the same function, but with no right shift. You can name any nondeleted message or list of messages.

6.6.7 Saving the Message in a File: `~w`

To save the current text of a message body in a file, use:

`~w filename`

Mail writes out the message body to the specified file, then prints the number of lines and characters written to the file. The `~w` escape does *not* write the message heading to the file.

6.6.8 Leaving Mail Temporarily: `~!` and `~|`

To temporarily escape to the shell, use the escape

~!command

This executes *command* and returns you to **mail** compose mode without altering your message. If you wish to filter the body of your message through a shell command, use

~!command

This pipes your message through the command and uses the output as the new text of your message. If the command produces no output, **mail** assumes that something is wrong, retains the old version of your message, and prints:

(continue)

6.6.9 Escaping to Mail Command Mode: ~:

To temporarily escape to **mail** command mode, use either of the escapes

~:mail-command

~_mail-command

You can then execute any **mail** command that you want. Note that this escape will not work in most cases if you enter compose mode from the XENIX shell. It depends on the command used (**set** and **unset** will work), but most commands that involve message lists are not allowed. You will receive the message:

May not execute *cmd* while composing

6.6.10 Placing a Tilde at the Beginning of a Line: ^^

If you wish to send a message that contains a line beginning with a tilde, you must type it twice. For example, typing

^^This line begins with a tilde.

appends

~This line begins with a tilde.

to your message. The escape character can be changed to a different character with the *escape* option. (For information on how to set options, see section 6.7, “Setting Up Your Environment: The .mailrc File.”) If the escape character is not a tilde, then this discussion applies to that

character and not the tilde.

6.7 Setting Up Your Environment: The .mailrc File

Whenever **mail** is invoked, it first reads the file */usr/lib/mail/mailrc* then the file *.mailrc* in the user's home directory. System-wide aliases are defined in */usr/lib/mail/mailrc*. Personal aliases and **set** options are defined in *.mailrc*. The following is a sample *.mailrc* file:

```
# number sign introduces comments

# personal aliases office and cohorts are defined below

alias office bill steve karen
alias cohorts john mary bob beth mike

# set dot lets messages be terminated by period on new line

# set askcc says to prompt for Cc: list after composing message

set dot askcc

# cd changes directory to different current directory

cd
```

6.7.1 The Subject Prompt: asksubject

The *asksubject* switch causes prompting for the subject of each message before you enter compose mode. If you respond to the prompt with a RETURN, then no subject field is sent.

6.7.2 The CC Prompt: askcc

The *askcc* switch causes prompting for additional carbon copy recipients when you finish composing a message. Responding with a RETURN signals your satisfaction with the current list. Pressing INTERRUPT prints

```
interrupt
(continue)
```

so that you can return to editing your message.

6.7.3 Printing the Next Message: autoprint

This switch causes the **delete** command to behave like **dp**. After deleting a message, the next message in the list is automatically printed. Printing also occurs automatically after execution of an **undelete** command.

6.7.4 Listing Messages in Chronological Order: chron and mchron

The *chron* switch causes messages to be listed in chronological order. By default, messages are listed with the most recent first. Set *chron* when you want to read a series of messages in the order they were received.

The *mchron* switch, like *chron*, prints messages in chronological order, but lists them in the opposite order, i.e. highest-numbered, or most recent, first. This is useful if you keep a large number of messages in your mailbox and you wish to list the headers of the most recently received mail first but read the messages themselves in chronological order.

6.7.5 Using the Period to Send a Message: *dot*

The *dot* switch lets you use a period (.) as an end-of-transmission character, as well as CNTRL-D. This option is available for those who are used to this convention when editing with the editor, *ed*.

6.7.6 Sending Mail While in Mail: *execmail*

It is often desirable to reply to a piece of mail, or send mail while reading your mail file. This process is speeded up by the use of the *execmail* option. It causes the underbar prompt to return before **mail** is finished being sent. This frees the user to continue while *mail* performs mailing functions in background. The *metoo* option will not work if *execmail* is set. *execmail* is set by default. Unset *execmail* and set *metoo* in the user's *.mailrc* file to use the *metoo* option.

6.7.7 Including Yourself in a Group: *metoo*

Usually, when a group is expanded that contains name of the sender, the sender is removed from the expansion. Setting the *metoo* option causes the sender to be included in the group.

6.7.8 Saving Aborted Messages: *save*

The *nosave* switch prevents aborted messages from being appended to the file *dead.letter* in your home directory; messages are saved by default. Messages are aborted when in compose mode by typing either two interrupts or a ~q compose escape.

6.7.9 Printing the Version Header: *quiet*

The *quiet* switch suppresses the printing of "<n> messages:" before the header-list and suppresses printing of the version header when **mail** is first invoked.

6.7.10 Choosing an Editor: The *EDITOR* String

The *EDITOR* string contains the pathname of the text editor to use in the **edit** command and ~e escape. If not defined, then the default editor is used. For example:

```
set EDITOR=/bin/ed
```

6.7.11 Choosing an Editor: The *VISUAL* String

The *VISUAL* string contains the pathname of the text editor used in the **visual** command and ~v escape. For example:

```
set VISUAL=/bin/vi
```

By default **vi** is the editor used.

6.7.12 Choosing a Shell: The SHELL String

The *SHELL* string contains the name of the shell to use in the **!** command and the **~!** escape. A default shell is used if this option is not defined. For example:

```
set SHELL=/bin/sh
```

6.7.13 Changing the Escape Character: The escape String

The *escape* string defines the character to use in place of the tilde (**~**) to denote compose escapes. For example:

```
set escape=*
```

With this setting, the asterisk becomes the new compose escape character.

6.7.14 Setting Page Size: The page String

The *page* string causes messages to be displayed in pages of size *n* lines. You are prompted with a question mark between pages. Pressing RETURN causes the next page of the current message to be printed. By default this paging feature is turned off.

6.7.15 Saving Outgoing Mail: The record String

The *record* string sets the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is not copied and saved. For example:

```
set record=/usr/john/recordfile
```

With this setting, all outgoing mail is automatically appended to the file */usr/john/recordfile*.

6.7.16 Keeping Mail in the System Mailbox: autombox

The *autombox* switch determines whether messages remain in the system mailbox when you exit **mail**. If you **set autombox**, examined messages are automatically placed in the *mbox* file in your home directory (your user mailbox) and *removed* from the system mailbox when you quit.

6.7.17 Changing the top Value: The toplines String

The *toplines* string sets the number of lines of a message to be printed out with the **top** command. By default, this value is five. For example:

```
set toplines=10
```

With this setting, ten lines of each message are printed out when the **top** command is used.

6.7.18 Sending Mail Over Telephone Lines: ignore

The *ignore* switch causes interrupt signals from your terminal to be ignored and echoed as at-signs (**@**). This switch is normally used only when communicating with **mail** over telephone lines.

6.8 Using Advanced Features

This section discusses advanced features of **mail** useful to those with some existing familiarity with the XENIX **mail** system.

6.8.1 Command Line Options

One very useful command line option to **mail** is the **-s** "subject" switch. With this switch you can specify a subject on the command line. For example, you could send a file named *letter* with the subject line, "Important Meeting at 12:00", by typing the following:

```
mail -s "Important Meeting at 12:00" john bob mike <letter
```

To include other header fields in your message, you can use the following options:

-b user Adds the blind carbon copy field to the message header.

-c user Adds the carbon copy field to the message header.

-r Adds the return-receipt to: field to the message header.

Mail also allows you to edit files of messages by using the **-f** switch on the command line. For example,

```
mail -f filename
```

causes **mail** to edit *filename* and

```
mail -f
```

causes **mail** to read *mbox* in your home directory. All the **mail** commands except **hold** are available to edit the messages. When you type the **quit** command, **mail** writes the updated file back.

If you send mail over a noisy phone line, you may notice that bad characters are transmitted. Many of these will be the character that aborts messages: the RUBOUT or DEL character. To ignore these bad characters, invoke **mail** with the **-i** switch.

When you enter the mail program (as opposed to sending a message from command level), two command line options are available:

-R Makes the mail session read-only, preventing alteration of the mail being read.

-u user Reads in *user's* mail instead of your own.

6.8.2 Using Mail as a Reminder Service

Besides sending and receiving mail, you can use **mail** as a reminder service. Several XENIX commands have this idea built in to them. For example, the XENIX **lpr** command's **-m** switch causes mail to be sent to the user after files have been printed on the lineprinter. XENIX automatically examines the file named *calendar* in each user's home directory and looks for lines containing either today or tomorrow's date. These lines are sent by **mail** as reminder of important events.

If you program in the shell command language, you can use **mail** to signal the completion of a job. For example, you might place the following two lines in a shell procedure:

```
biglongjob
echo "biglongjob done" | mail self
```


You can also create a logfile that you want to mail to yourself. For example, you might have a shell procedure that looks like this:

```
dosomething >logfile  
mail self <logfile
```

For information about writing shell procedures, see Chapter 7 of this manual, "The Shell."

6.8.3 Handling Large Amounts of Mail

Eventually, you will face the problem of dealing with an accumulation of messages in your user mailbox. There are a number of strategies that you can employ to handle this flood of information. Keep in mind the dictum:

When in doubt, throw it out.

This means that you should only save *important* mail in your user mailbox. If your mailbox file becomes large, you must periodically examine its contents to decide whether messages are still relevant. For very long messages, consider replacing message contents with summaries.

Even the above measures are not usually help enough in organizing the many messages you are likely to receive. One effective approach is to save mail in files organized by sender, by topic, or by a combination of the two. Create these files in a separate **mail** directory; you can access these mailbox files with the **mail -f filename** switch. However, be forewarned—this approach to organizing mail quickly eats up disk space.

6.8.4 Maintenance and Administration

The following is a list of the programs and files that make up the XENIX **mail** system:

/usr/bin/mail	Mail program
/usr/lib/mail/mailrc	Mail system initialization file
/usr/spool/mail/*	System mailbox files
/usr/name/dead.letter	File where undeliverable mail is deposited
/usr/name/mbox	User mailbox
/usr/name/.mailrc	User mail initialization file
/usr/lib/mail/mailhelp.cmd	Mail command help file
/usr/lib/mail/mailhelp.esc	Mail compose escape help file
/usr/lib/mail/mailhelp.set	Mail option help file
/usr/lib/mail/aliases	System-wide aliases
/usr/lib/mail/aliases.hash	System-wide alias database
/usr/lib/mail/faliases	Forwarding aliases
/usr/lib/mail/maliases	Machine aliases

A system-wide distribution list is kept in */usr/lib/mail/aliases*. A system administrator is usually in charge of this list. These aliases are kept in a vastly different syntax from *.mailrc*, and

are expanded when mail is sent. You will normally need special permission to change system-wide aliases.

6.9 Quick Reference

The following sections provide quick reference to the available commands, compose escapes, and options.

6.9.1 Command Summary

Given below are the name and syntax for each command, its abbreviated form (in brackets), and a short description. Many commands have optional arguments; most can be executed without any arguments at all. In particular, commands that take a message-list argument default will to the current message if no message-list is given. In the following descriptions, boldface denotes the name of a command, compose escape or option. Italics are used for arguments to commands or compose escapes. The vertical bar indicates selection and is used to separate the arguments from which you may select. All other text should be read literally.

RETURN	Prints the next message.
+<i>n</i>	[+] With no <i>n</i> argument, goes to the next message and prints it. If given a numeric argument <i>n</i> , goes to the <i>n</i> th message and prints it.
-<i>n</i>	[-] With no <i>n</i> argument, goes to the previous message and prints it. If given a numeric argument <i>n</i> , goes to the <i>n</i> th previous message and prints it.
^	Prints the first message.
\$	Prints the last message.
=	Prints the message number of the current message.
?	Prints the summary of mail commands in <i>/usr/lib/mail/mailhelp.cmd</i> .
!shell-cmd	Executes the shell command that follows. No space is needed after the exclamation point.
Alias <i>users</i>	Prints system-wide aliases for users. At least one user must be specified.
alias <i>name users</i>	[a] Aliases <i>users</i> to <i>name</i> . With no name arguments, prints all currently defined aliases. With one argument, prints the users aliased by the given name argument.
cd <i>directory</i>	[c] Changes the user's working directory to the specified directory. If no directory is given, then changes to the user's home directory.
delete <i>mesg-list</i>	[d] Deletes each message in the given message-list.
dp <i>mesg-list</i>	Deletes the current message and prints the next message.
echo <i>path</i>	Expands shell metacharacters.
edit <i>mesg-list</i>	[e] Takes the given message-list and points the text editor at each message in turn. On return to command mode, the edited message is read

back in. See also the **visual** command.

exit[!] [x] Immediately returns to the shell without modifying the system mailbox, the user mailbox, or a file specified with the **-f** switch.

file [fi] Prints the name of the mailbox file.

forward *mesg-num user-list*

[f] Takes a *user-list* argument and forwards the current message to each name. The message sent to each is indented and shows that the sender has passed it on. The *mesg-num* argument is optional, and is used to forward the numbered message instead of the default message.

Forward *mesg-num user-list*

[F] Same as **forward** except that the message is not indented.

headers *+n|-n|mesg-list*

[h] With no argument, lists the current range of headers, which is an 18-message group. If a plus (+) argument is given, then the next 18-message group is printed, and if a minus (-) argument is given, the previous 18-message group is printed. Both plus and minus accept an optional numeric argument indicating the number of header-windows to move forward or backward. If a message-list is given, then the message-header for each message in the list is printed.

help Same as ? above. Prints the summary of **mail** commands in */usr/lib/mail/mailhelp.cmd*.

hold *mesg-list* [ho] Takes a message-list and marks each message to be saved in the user's system mailbox instead of in *mbox*.

list Prints list of **mail** commands.

lpr *mesg-list* [l] Prints each of the messages in the required message-list on the lineprinter. Messages are piped through *pr* before being printed.

mail [*user-list*] [m] Takes an optional user-list argument and sends mail to each name after entering compose mode.

mbox *mesg-list* [mb] Marks messages given in the message-list argument to be saved in the user mailbox when a **quit** is executed. Message headers contain an initial letter "M" to show that they are to be saved.

move *mesg-list mesg-num*

Places the messages specified in *mesg-list* after the message specified in *mesg-num*. If *mesg-num* is 0, *mesg-list* moves to the top of the mailbox.

print *mesg-list* [p] Takes a message-list and prints each message on the user's terminal.

quit [q] Terminates the **mail** session, retaining all nondeleted, unsaved messages in the system mailbox. If the *autombox* option is set, then examined messages are saved in the user mailbox, deleted messages are discarded, and all messages marked with the **hold** command are retained in the system mailbox.

If you are executing a **quit** while editing a mailbox file with the **-f** flag, the mailbox file is rewritten and the user returns to the shell.

reply <i>mesg-list</i>	[r] Takes a message-list and sends mail to each message author just like the mail command.
Reply <i>mesg-list</i>	[R] Sends a reply to users named in the <i>To:</i> and <i>Cc:</i> fields, as well as the original sender.
restart	Reads in mail that arrives during the current mail session.
save <i>mesg-list filename</i>	[s] Takes an optional message-list and a filename and appends each message in turn to the end of the file. The default message is the current message.
set	[se] Prints list of available options.
set <i>option-list</i>	[se] With no arguments, prints all variable values. Otherwise, sets option. Arguments are of the form <i>option=value</i> , if the option is a string option or just <i>option</i> , if the option is a switch. Multiple options may be set on one line.
shell	[sh] Invokes an interactive version of the shell.
size <i>mesg-list</i>	[si] Takes a message-list and prints the size in characters of each message.
source <i>file</i>	[so] Reads and executes mail commands from the given file.
string <i>string mesg-list</i>	Searches for <i>string</i> in <i>mesg-list</i> . If no <i>mesg-list</i> is specified, all undeleted messages are searched. Ignores case in search.
top	[t] Takes a message-list and prints the top five lines. The number of lines printed is set by the variable <i>toplines</i> .
undelete <i>mesg-list</i>	[u] Takes a message-list and marks each one as <i>not</i> being deleted. Each message in the list must previously have been deleted.
unset <i>options</i>	[uns] Takes a list of option names and discards their remembered values; this is the opposite of set .
visual <i>mesg-list</i>	[v] Takes a message-list and invokes the vi editor on each one.
whois	Looks up a list of target mail recipients and prints the real names or descriptions of each recipient. If the first character of the first argument is alphabetic, the arguments are looked up without change. Otherwise, the arguments are assumed to be a message list, in the format specified in the <i>Mail User's Guide</i> . For each message in the list, the "From" person is extracted from the header and added to list of users to be searched.
write <i>mesg-list filename</i>	[w] Writes the message bodies of messages given by the message-list to the file given by <i>filename</i> .

6.9.2 Compose Escape Summary

Compose escapes are used when composing messages to perform special functions. They are only recognized at the beginning of lines. The escape character can be set with the *escape* string option. (See section 6.7.14, "The escape String.") Abbreviations for each escape are in brackets.

Here is a summary of the compose escapes:

<code>~~string</code>	Inserts the string of text in the message prefaced by a single tilde (~).
<code>~?</code>	Prints out help for compose escapes on terminal.
<code>~.</code>	Same as CNTRL-D on a new line.
<code>~!command</code>	Executes a shell command, then returns to compose mode.
<code>~ command</code>	Pipes the message body through the command as a filter. Replaces the message body with the output of the filter. If the command gives no output or terminates abnormally, retains the original message body.
<code>~_mail-command</code>	Executes a mail command, then returns to compose mode.
<code>~:mail-command</code>	Executes a mail command, then returns to compose mode.
<code>~alias</code>	[~a] Prints list of private aliases.
<code>~alias aliasname</code>	[~a] Prints names included in private <i>aliasname</i> .
<code>~alias aliasname users</code>	[~a] Adds <i>users</i> to private <i>aliasname</i> list.
<code>~Alias</code>	[~A] Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). The user list is taken from header fields.
<code>~Alias users</code>	[~A] Performs aliasing by first examining private aliases and then system-wide aliases using all three global alias files. Only the final result is printed (non-local mail recipients will have the complete delivery path printed). At least one user must be specified.
<code>~bcc name ...</code>	[~b] Adds the given names to the <i>Bcc:</i> field.
<code>~cc name ...</code>	[~c] Adds the given name to the <i>cc:</i> field.
<code>~dead</code>	[~d] Reads the file <i>dead.letter</i> from your home directory into the message.
<code>~editor</code>	[~e] Invokes the line editor on the message being sent. Exiting the editor returns the user to compose mode.
<code>~headers</code>	[~h] Edits the message heading fields by printing each one in turn and allowing the user to modify each field.
<code>~message mesg-list</code>	[~m] Reads the named messages into the message being sent, shifted right one tab. If no messages are specified, reads the current message.

~Message <i>mesg-list</i>	[~M] Same as ~message except with no right shift.
~print	[~p] Prints the message buffer prefaced by the message heading.
~Print	[~P] Prints the real names or descriptions (in parentheses) after each recipient.
~quit	[~q] Aborts the message being sent, copying the message to <i>dead.letter</i> in your home directory if the <i>save</i> option is set.
~read <i>filename</i>	[~r] Reads the named file into the message.
~Return <i>name</i>	[~R] Adds the given names to the <i>Return-receipt-to:</i> field.
~shell	[~sh] Invokes a shell.
~subject <i>string</i>	[~s] Causes the named string to become the current subject field.
~to <i>name ...</i>	[~t] Adds the given names to the <i>To:</i> field.
~visual	[~v] Invokes the vi editor to edit the message buffer. Exiting the editor returns the user to compose mode.
~write <i>filename</i>	[~w] Writes the message body to the named file.

6.9.3 Option Summary

Options are controlled with the **set** and **unset** commands. An option is either a switch or a string. A switch is either on or off, while a string option has a value that is a pathname, a number, or a single character. Options are summarized below.

askcc	Causes prompting for additional carbon copy recipients at the end of each message. Pressing RETURN retains the current list.
asksubject	Causes prompting for the subject of each message you send. The subject is a line of text terminated by a RETURN.
autombox	Usually messages are retained in the system mailbox when the user quits. However, if this option is set , examined messages are automatically appended to the user mailbox.
autoprint	Causes the delete command to behave like dp . Thus, after deleting (or undeleting) a message, the next one is printed automatically.
chron	Causes messages to be listed in chronological order.
dot	Causes a single period on a newline to act as the EOT character. The normal end-of-transmission character, CNTRL-D, still works.
EDITOR=	Pathname of the text editor to use in the edit command and ~e escape. If not defined, then a default editor is used.
escape <i>char</i>	If defined, sets <i>char</i> as the character to use in place of the tilde (~) to denote compose escapes.

XENIX User's Guide

execmail	Causes the underbar prompt to return before <i>mail</i> is finished being sent.
ignore	Causes interrupt signals from your terminal to be ignored and echoed as at-signs (@).
mchron	Causes messages to be listed in numerical order (most recently received first), but displayed in chronological order.
metoo	Normally, before sending, the name of the sender is removed from alias expansions. If <i>metoo</i> is set, then the name of the sender is <i>not</i> removed.
nosave	Prevents saving of the message buffer in the file <i>dead.letter</i> in the home directory, after two consecutive interrupts or a ~q escape.
page=<i>n</i>	Specifies the number of lines (<i>n</i>) to be printed in a "page" of text when displaying messages.
quiet	Suppresses the printing of the version when mail is first invoked.
record=	Sets the pathname of the file used to record all outgoing mail. If not defined, then outgoing mail is <i>not</i> copied.
SHELL=	Pathname of the shell to use in the ! command and the ~! escape. A default shell is used if this option is not defined.
toplines=	Sets the number of lines of a message to be printed with the top command. Default is five lines.
verify	Causes each target mail recipient to be verified. This option permits errors made while composing messages to be corrected or ignored.
VISUAL=	Pathname of the text editor to use in the visual command and ~v escape. The default is for the vi editor.

Chapter 7

The Shell

7.1	Introduction	1
7.2	Basic Concepts	1
7.2.1	How Shells Are Created	1
7.2.2	Commands	1
7.2.3	How the Shell Finds Commands	2
7.2.4	Generation of Argument Lists	2
7.2.5	Quoting Mechanisms	3
7.3	Redirecting Input and Output	4
7.3.1	Standard Input and Output	4
7.3.2	Diagnostic and Other Outputs	4
7.3.3	Command Lines and Pipelines	5
7.3.4	Command Substitution	6
7.4	Shell Variables	7
7.4.1	Positional Parameters	7
7.4.2	User-Defined Variables	7
7.4.3	Predefined Special Variables	9
7.5	The Shell State	10
7.5.1	Changing Directories	10
7.5.2	The .profile File	10
7.5.3	Execution Flags	11
7.6	A Command's Environment	11
7.7	Invoking the Shell	12
7.8	Passing Arguments to Shell Procedures	12
7.9	Controlling the Flow of Control	13
7.9.1	Using the if Statement	14
7.9.2	Using the case Statement	15
7.9.3	Conditional Looping: while and until	16
7.9.4	Looping Over a List: for	16
7.9.5	Loop Control: break and continue	17
7.9.6	End-of-File and exit	18
7.9.7	Command Grouping: Parentheses and Braces	18
7.9.8	Input/Output Redirection and Control Commands	19
7.9.9	Transfer to Another File and Back: The Dot (.) Command	19
7.9.10	Interrupt Handling: trap	19
7.10	Special Shell Commands	21
7.11	Creation and Organization of Shell Procedures	22
7.12	More About Execution Flags	23

7.13	Supporting Commands and Features	24
7.13.1	Conditional Evaluation: test	24
7.13.2	Echoing Arguments	25
7.13.3	Expression Evaluation: expr	26
7.13.4	True and False	26
7.13.5	In-Line Input Documents	26
7.13.6	Input/Output Redirection Using File Descriptors	27
7.13.7	Conditional Substitution	27
7.13.8	Invocation Flags	28
7.14	Effective and Efficient Shell Programming	29
7.14.1	Number of Processes Generated	29
7.14.2	Number of Data Bytes Accessed	30
7.14.3	Shortening Directory Searches	30
7.14.4	Directory-Search Order and the PATH Variable	31
7.14.5	Good Ways to Set Up Directories	31
7.15	Shell Procedure Examples	31
7.16	Shell Grammar	38

7.1 Introduction

When users log into XENIX, they communicate with the shell command interpreter, **sh**. This interpreter is a XENIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one function: to read and execute commands from its standard input.

Because the shell gives the user a high-level language in which to communicate with the operating system, XENIX can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With XENIX and the shell, commands can be:

- Combined to form new commands
- Passed positional parameters
- Added or renamed by the user
- Executed within loops or executed conditionally
- Created for local execution without fear of name conflict with other user commands
- Executed in the background without interrupting a session at a terminal

Furthermore, commands can “redirect” command input from one source to another and redirect command output to a file, terminal, printer, or to another command. This provides flexibility in tailoring a task for a particular purpose.

7.2 Basic Concepts

The shell itself (i.e., the program that reads your commands when you log in or that is invoked with the **sh** command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

7.2.1 How Shells Are Created

In XENIX, a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and may even start (or “fork”) new processes. Thus, at any given moment several processes may be executing, some of which are “children” of other processes.

Users log into the operating system and are assigned a “shell” from which they execute. This shell is a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In the XENIX multitasking environment, files may be created in one phase and then sent off to be processed in the “background.” This allows the user to continue working while programs are running.

7.2.2 Commands

The most common way of using the shell is by typing simple commands at your keyboard. A *simple command* is any sequence of arguments separated by spaces or tabs. The first argument (numbered zero) specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. For example, the following command line might be typed to request printing of the files *allan*, *barry*, and *calvin*:

```
lpr allan barry calvin
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell, as

parent, creates a child process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a *subshell*) to read the file and execute the commands inside it.

From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This provides uniformity of invocation.

7.2.3 How the Shell Finds Commands

The shell normally searches for commands in three distinct locations in the file system. The shell attempts to use the command name as given; if this fails, it prepends the string */bin* to the name. If the latter is unsuccessful, it prepends */usr/bin* to the command name. The effect is to search, in order, the current directory, then the directory */bin*, and finally, */usr/bin*. For example, the **pr** and **man** commands are actually the files */bin/pr* and */usr/bin/man*, respectively. A more complex pathname may be given, either to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If a given command name begins with a slash (/) (e.g., */bin/sort* or */cmd*), the prepending is not performed. Instead, a single attempt is made to execute the command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the shell *PATH* variable. (Shell variables are discussed later in this chapter).

7.2.4 Generation of Argument Lists

The arguments to commands are very often filenames. Sometimes, these filenames have similar, but not identical, names. To take advantage of this similarity in names, the shell lets the user specify patterns that match the filenames in a directory. If a pattern is matched by one or more filenames in a directory, then those filenames are automatically generated by the shell as arguments to the command.

Most characters in such a pattern match themselves, but there are also XENIX special characters that may be included in a pattern. These special characters are: the star (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters. Inside brackets, a pair of characters separated by a dash (—) matches any character within the range of that pair. Thus *[a—de]* is equivalent to *[abcde]*.

Examples of metacharacter usage:

<i>*</i>	<i>(Matches all names in the current directory)</i>
<i>*temp*</i>	<i>(Matches all names containing "temp")</i>
<i>[a—f]*</i>	<i>(Matches all names beginning with "a" through "f")</i>
<i>*.c</i>	<i>(Matches all names ending in ".c")</i>
<i>/usr/bin/?</i>	<i>(Matches all single-character names in /usr/bin)</i>

This pattern-matching capability saves typing and, more importantly, makes it possible to organize information in large collections of files that are named in a structured fashion, using common characters or extensions to identify related files.

Pattern matching has some restrictions. If the first character of a filename is a period (.), it can be matched only by an argument that literally begins with a period. If a pattern does not match any

filenames, then the pattern itself is printed out as the result of the match.

Note that directory names should not contain any of the following characters:

* ? []

If these characters are used, then infinite recursion may occur during pattern matching attempts.

7.2.5 Quoting Mechanisms

The characters `<`, `>`, `*`, `?`, `|` and `G` have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. This is done by using single quotation marks (`'`) or double quotation marks (`"`) to surround a string. A backslash (`\`) before a single character provides this function. (Back quotation marks (```) are used only for command substitution in the shell and do not hide the special meanings of any characters.)

All characters within single quotation marks are taken literally. Thus

```
echostuff='echo $? $*; ls * | wc'
```

results in the string

```
echo $? $*; ls * | wc
```

being assigned to the variable *echostuff*, but it does *not* result in any other commands being executed.

Within double quotation marks, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign (`$`), the backslash (`\`), the single quotation mark (`'`), and the double quotation mark (`"`) itself. Thus, within double quotation marks, variables are expanded and command substitution takes place (both topics are discussed in later sections). However, any commands in a command substitution are unaffected by double quotation marks, so that characters such as star (`*`) retain their special meaning.

To hide the special meaning of the dollar sign (`$`) and single and double quotation marks within double quotation marks, precede these characters with a backslash (`\`). Outside of double quotation marks, preceding a character with a backslash is equivalent to placing single quotation marks around that character. A backslash (`\`) followed by a newline causes that newline to be ignored and is equivalent to a space. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

Some examples of quoting are shown below :

Input	Shell interprets as:
<code>'`'</code>	The back quotation mark (<code>`</code>)
<code>'"'</code>	The double quotation mark (<code>"</code>)
<code>`echo one`</code>	the one word <code>"`echo one`"</code>
<code>"\""</code>	The double quotation mark (<code>"</code>)
<code>"`echo one`"</code>	the one word <code>"one"</code>
<code>""</code>	illegal (expects another <code>`</code>)
<code>one two</code>	the two words <code>"one"</code> & <code>"two"</code>
<code>"one two"</code>	the one word <code>"one two"</code>
<code>'one two'</code>	the one word <code>"one two"</code>
<code>'one * two'</code>	the one word <code>"one * two"</code>
<code>"one * two"</code>	the one word <code>"one * two"</code>
<code>`echo one`</code>	the one word <code>"one"</code>

7.3 Redirecting Input and Output

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

7.3.1 Standard Input and Output

When a command begins execution, it usually expects that three files are already open: a “standard input”, a “standard output”, and a “diagnostic output” (also called “standard error”). A number called a *file descriptor* is associated with each of these files. By convention, file descriptor 0 is associated with the standard input, file descriptor 1 with the standard output, and file descriptor 2 with the diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the terminal screen). The shell permits the files to be redirected elsewhere before control is passed to an invoked command.

An argument to the shell of the form “<*file*” or “>*file*” opens the specified file as the standard input or output (in the case of output, destroying the previous contents of *file*, if any). An argument of the form “>>*file*” directs the standard output to the end of *file*, thus providing a way to append data to the file without destroying its existing contents. In either of the two output cases, the shell creates *file* if it does not already exist. Thus

```
> output
```

alone on a line creates a zero-length file. The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of filenames occurs after these substitutions. This means that

```
echo 'this is a test' > *.gal
```

produces a one-line file named **.gal*. Similarly, an error message is produced by the following command, unless you have a file with the name “?”:

```
cat < ?
```

So remember, special characters are *not* expanded in redirection arguments. The reason this is so is that redirection arguments are scanned by the shell *before* pattern recognition and expansion takes place.

7.3.2 Diagnostic and Other Outputs

Diagnostic output from XENIX commands is normally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines.) You can redirect this error output to a file by immediately prepending the number of the file descriptor (2 in this case) to either output redirection symbol (> or >>). The following line appends error messages from the *cc* command to the file named *ERRORS*:

```
cc testfile.c 2>>ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening spaces or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if *cmd* puts output on file descriptor 9, then the following line will direct that output to the file *savedata*:

```
cmd 9>savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose, for example, that *cmd* directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard 2>error 9>data
```

7.3.3 Command Lines and Pipelines

A sequence of commands separated by the vertical bar (|) makes up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by *pipes*, that is, the output of each command (except the last one) becomes the input of the next command in line.

A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; *sort* is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline:

```
nroff -mm text | col | lpr
```

Nroff is a text formatter available in the XENIX Text Processing System whose output may contain reverse line motions, **col** converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and **lpr** does the actual printing. The flag **-mm** indicates one of the commonly used formatting options, and *text* is the name of the file to be formatted.

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these at a terminal:

- **who**
Prints the list of logged-in users on the terminal screen.
- **who >>log**
Appends the list of logged-in users to the end of file *log*.
- **who|wc -l**
Prints the number of logged-in users. (The argument to **wc** is pronounced “minus ell”.)
- **who|pr**
Prints a paginated list of logged-in users.
- **who|sort**
Prints an alphabetized list of logged-in users.
- **who|grep bob**
Prints the list of logged-in users whose login names contain the string *bob*.

- `who|grep bob|sort|pr`
Prints an alphabetized, paginated list of logged-in users whose login names contain the string *bob*.
- `{ date; who|wc -l; } >>log`
Appends (to file *log*) the current date followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.
- `who|sed -e 's/ .*//'|sort|uniq -d`
Prints only the login names of all users who are logged in more than once. Note the use of **sed** as a filter to remove characters trailing the login name from each line. (The “.” in the **sed** command is preceded by a space.)

The **who** command does not *by itself* provide options to yield all these results—they are obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace “`who|`” with “`</etc/passwd`” in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line, even at the start. This means that

```
<infile >outfile sort|pr
```

is the same as

```
sort|pr <infile >outfile
```

7.3.4 Command Substitution

Any command line can be placed within back quotation marks (``...``) so that the output of the command replaces the quoted command line itself. This concept is known as *command substitution*. The command or commands enclosed between back quotation marks are first executed by the shell and then their output replaces the whole expression, back quotation marks and all. This feature is often used to assign to shell variables. (Shell variables are described in the next section.) For example,

```
today=`date`
```

assigns the string representing the current date to the variable “today”; for example “Tue Nov 27 16:01:09 EST 1982”. The following command saves the number of logged-in users in the shell variable *users*:

```
users=`who | wc -l`
```

Any command that writes to the standard output can be enclosed in back quotation marks. Back quotation marks may be nested, but the inside sets must be escaped with backslashes (`\`). For example:

```
logmsg=`echo Your login directory is \`pwd\``
```

will display the line “your login directory is *name of login directory*”. Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example,

```
read first init last
```

takes an input line of the form

```
G. A. Snyder
```

and has the same effect as typing:

```
first=G.  init=A.  last=Snyder
```

The **read** command assigns any excess “words” to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

7.4 Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as *positional parameters*; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself may assign string values.

7.4.1 Positional Parameters

When a shell procedure is invoked, the shell implicitly creates *positional parameters*. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter \$0. The first command argument is called \$1, and so on. The **shift** command may be used to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files:

```
while test '$1'
do case $1 in
    -a) A=aooption ; shift ;;
    -b) B=booption ; shift ;;
    -c) C=cooption ; shift ;;
    -*) echo "bad option" ; exit 1 ;;
    *) process rest of files
    esac
done
```

One can explicitly force values into these positional parameters by using the **set** command. For example,

```
set abc def ghi
```

assigns the string “abc” to the first positional parameter, \$1, the string “def” to \$2, and the string “ghi” to \$3. Note that \$0 may not be assigned a value in this way—it always refers to the name of the shell procedure; or in the login shell, to the name of the shell.

7.4.2 User-Defined Variables

The shell also recognizes alphanumeric variables to which string values may be assigned. A simple assignment has the syntax:

```
name=string
```

Thereafter, *\$name* will yield the value *string*. A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. No spaces surround the equal sign (=) in an assignment statement. Note that positional parameters may not appear on the left side of an assignment statement; they can only be set as described in the previous section.

More than one assignment may appear in an assignment statement, but beware: *the shell performs the assignments from right to left*. Thus, the following command line results in the variable “A” acquiring the value “abc”:


```
A=$B B=abc
```

The following are examples of simple assignments. Double quotation marks around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, while also allowing variable substitution (also known as “parameter substitution”) to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign (\$) are replaced by the corresponding values, if any. Single quotation marks inhibit variable substitution:

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable “echovar” has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string “echo”. No quotation marks are needed around the string of asterisks being assigned to *stars* because pattern matching (expansion of star, the question mark, and brackets) does not apply in this context. Note that the value of *\$asterisks* is the literal string “\$stars”, *not* the string “*****”, because the single quotation marks inhibit substitution.

In assignments, spaces are not re-interpreted after variable substitution, so that the following example results in *\$first* and *\$second* having the same value:

```
first='a string with embedded spaces'
second=$first
```

In accessing the values of variables, you may enclose the variable name in braces {...} to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input:

```
a='This is a string'
echo "${a}ent test of variables."
```

Here, the **echo** command prints:

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for “\$aent” and print:

```
test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user:

HOME Initialized by the **login** program to the name of the user's *login directory*, that is, the directory that becomes the current directory upon completion of a login; **cd** without arguments switches to the \$HOME directory. Using this variable helps keep full pathnames out of shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.

IFS The variable that specifies which characters are *internal field separators*. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set IFS to include that delimiter.) The shell initially sets IFS to include the blank, tab, and newline characters.

MAIL The pathname of a file where your mail is deposited. If MAIL is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). MAIL must be set by the user and “exported”. (The **export** command is discussed later in this chapter.) (The presence of mail in the standard mail file is

also announced at login, regardless of whether MAIL is set.)

PATH The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes PATH to the list `:/bin:/usr/bin` where a null argument appears in front of the first colon. A null anywhere in the path list represents the current directory. On some systems, a search of the current directory is *not* the default and the PATH variable is initialized instead to `/bin:/usr/bin`. If you wish to search your current directory last, rather than first, use:

```
PATH=/bin:/usr/bin::
```

Here, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, `$HOME/bin`) and cause it to be searched *before* the other three directories by using:

```
PATH=$HOME/bin::/bin:/usr/bin
```

“PATH” is normally set in your `.profile` file.

PS1 The variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is “\$ ” (a dollar sign (\$) followed by a blank).

PS2 The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of PS2. The default value for this variable is “> ” (a greater-than symbol followed by a space).

In general, you should be sure to **export** all of the above variables so that their values are passed to all shells created from your login. Use **export** at the end of your `.profile` file. An example of an **export** statement follows:

```
export HOME IFS MAIL PATH PS1 PS2
```

7.4.3 Predefined Special Variables

Several variables have special meanings; the following are set *only* by the shell:

\$# Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, `$#` yields the number of the highest set positional parameter. Thus

```
sh cmd a b c
```

automatically sets `$#` to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
    echo two or more args required; exit
fi
```

\$? Contains the exit status of the last command executed (also referred to as “return code”, “exit code”, or “value”). Its value is a decimal string. Most XENIX commands return zero to indicate successful completion. The shell itself returns the current value of `$?` as its exit status.

- \$\$** The process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files. XENIX provides no mechanism for the automatic creation and deletion of temporary files; a file exists until it is explicitly removed. Temporary files are generally undesirable objects; the XENIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files; note that the directories */usr* and */usr/tmp* are cleared out if the system is rebooted.

```
#          use current process id
#          to form unique temp file
temp=/usr/temp/$$
ls > $temp
#          commands here, some of which use $temp
rm $temp
#          clean up at end
```

- \$!** The process number of the last process run in the background (using the ampersand (&)). This is a string containing from one to five digits.
- \$-** A string consisting of names of execution flags currently turned on in the shell. For example, **\$-** might have the value "xv" if you are tracing your output.

7.5 The Shell State

The state of a given instance of the shell includes the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory.

The state of a shell may be altered in various ways. These include changing the working directory with the **cd** command, setting several flags, and by reading commands from the special file, *.profile*, in your login directory.

7.5.1 Changing Directories

The **cd** command changes the current directory to the one specified as its argument. This can and should be used to change to a convenient place in the directory structure. Note that **cd** is often placed within parentheses to cause a subshell to change to a different directory and execute some commands without affecting the original shell.

For example, the first sequence below copies the file */etc/passwd* to */usr/you/passwd*; the second example first changes directory to */etc* and then copies the file:

```
cp /etc/passwd /usr/you/passwd
(cd /etc ; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

7.5.2 The *.profile* File

The file named *.profile* is read each time you log in to XENIX. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from *.profile*, does the shell read commands from the standard input—usually the terminal.

7.5.3 Execution Flags

The **set** command lets you alter the behavior of the shell by setting certain shell flags. In particular, the **-x** and **-v** flags may be useful when invoking the shell as a command from the terminal. The flags **-x** and **-v** may be set by typing:

```
set -xv
```

The same flags may be turned *off* by typing:

```
set +xv
```

These two flags have the following meaning:

- v** Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.
- x** Commands and their arguments are printed as they are executed. (Shell control commands, such as **for**, **while**, etc., are not printed, however.) Note that **-x** causes a trace of only those commands that are actually executed, whereas **-v** prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within shell procedures.

7.6 A Command's Environment

All variables and their associated values that are known to a command at the beginning of its execution make up its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
#          keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are *not* counted as arguments to the procedure and do not affect \$#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, type:

```
export
```

You will also get a list of variables that have been made **readonly**. To get a list of name-value

pairs in the current environment, type either

`printenv`

or

`env`

7.7 Invoking the Shell

The shell is a command and may be invoked in the same way as any other command:

`sh proc [arg...]` A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated.

`sh -v proc [arg...]` This is equivalent to putting “set -v” at the beginning of *proc*. It can be used in the same way for the -x, -e, -u, and -n flags.

`proc [arg...]` If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of:

`sh proc args`

An advantage of this form is that variables that have been exported in the shell will still be exported from *proc* when this form is used (because the shell only forks to read commands from *proc*). Thus any changes made within *proc* to the values of exported variables will be passed on to subsequent commands invoked from *proc*.

7.8 Passing Arguments to Shell Procedures

When a command line is scanned, any character sequence of the form `$n` is replaced by the *n*th argument to the shell, counting the name of the shell procedure itself as \$0. This notation permits direct reference to the procedure name and to as many as nine positional parameters. Additional arguments can be processed using the **shift** command or by using a **for** loop.

The **shift** command shifts arguments to the left; i.e., the value of \$1 is thrown away, \$2 replaces \$1, \$3 replaces \$2, and so on. The highest-numbered positional parameter becomes *unset* (\$0 is never shifted). For example, in the shell procedure *ripple* below, **echo** writes its arguments to the standard output.

```
#           ripple command
while test $# != 0
do
    echo $1 $2 $3 $4 $5 $6 $7 $8 $9
    shift
done
```

Lines that begin with a number sign (#) are comments. The looping command, **while**, is discussed in Section 7.9.3 of this chapter. If the procedure were invoked with

`ripple a b c`

it would print:

```
a b c
b c
c
```

The special shell variable “star” (`$*`) causes substitution of all positional parameters except `$0`. Thus, the `echo` line in the *ripple* example above could be written more compactly as:

```
echo $*
```

These two `echo` commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The shell star variable (`$*`) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command: For example

```
wc $*
```

counts the words of each of the files named on the command line.

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a newline or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via back quotation marks) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for *internal field separators*, that is, for any characters specified by IFS to break the command line into distinct arguments; *explicit* null arguments (specified by `"` or `'`) are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then filename generation occurs with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes, command lines are built inside a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command `eval` is available for this purpose. `Eval` takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output='| wc -l'
eval $command $output
```

This segment of code results in the execution of the command line

```
who | wc -l
```

The output of `eval` cannot be redirected. However, uses of `eval` can be nested, so that a command line can be evaluated several times.

7.9 Controlling the Flow of Control

The shell provides several commands that implement a variety of control structures useful in controlling the flow of control in shell procedures. Before describing these structures, a few terms need to be defined.

A *simple command* is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, *not* to the command.

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by vertical bars (`|`). In a pipeline, the standard output of each command but the last is connected (by a *pipe*) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is nonzero if the exit status of either the first or last process in the pipeline is nonzero.

A *command list* is a sequence of one or more pipelines separated by a semicolon (`;`), an ampersand (`&`), an “and-if” symbol (`&&`), or an “or-if” (`||`) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous

pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing INTERRUPT or QUIT. It is also immune to logouts with CNTRL-D. However, CNTRL-D *will* abort the command if you are operating over a dial-up line. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. In the above example without **nohup**, if you log out from a dial-up line while **cc** is still executing, **cc** will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of background processes without a compelling reason for doing so.

The and-if and or-if (&& and ||) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are lower than the ampersand (&)) and the vertical bar (|). In the command line

```
cmd1 | cmd2
```

the first command, *cmd1*, is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:

```
if          cmd1
            test $? != 0
then
            cmd2
fi
```

The and-if operator (&&) operator yields a complementary test. For example, in the following command line

```
cmd1 && cmd2
```

the second command is executed only if the first *succeeds* (and has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents:

```
{ nroff -mm text1; nroff -mm text2; } |lpr
```

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

7.9.1 Using the if Statement

The shell provides structured conditional capability with the **if** command. The simplest **if** command has the following form:

```

if command-list
then command-list
fi

```

The command list following the **if** is executed and if the last command in the list has a zero exit status, then the command list that follows **then** is executed. The word **fi** indicates the end of the **if** command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an **else** clause can be given with the following structure:

```

if command-list
then command-list
else command-list
fi

```

Multiple tests can be achieved in an **if** command by using the **elif** clause, although the **case** statement (See Section 7.9.2) is better for large numbers of tests. For example:

```

if          test -f "$1"
#                               is $1 a file?
then        pr $1
elif       test -d "$1"
#                               else, is $1 a directory?
then        (cd $1; pr *)
else        echo $1 is neither a file nor a directory
fi

```

The above example is executed as follows: if the value of the first positional parameter is a filename (-f), then print that file; if not, then check to see if it is the name of a directory (-d). If so, change to that directory (**cd**) and print all the files there (**pr***). Otherwise, **echo** the error message.

The **if** command may be nested (but be sure to end each one with a **fi**). The newlines in the above examples of **if** may be replaced by semicolons.

The exit status of the **if** command is the exit status of the last command executed in any **then** clause or **else** clause. If no such command was executed, **if** returns a zero exit status.

Note that an alternate notation for the **test** command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

```

if          [ -f "$1" ]
#                               is $1 a file?
then        pr $1
elif       [ -d "$1" ]
#                               else, is $1 a directory?
then        (cd $1; pr *)
else        echo $1 is neither a file nor a directory
fi

```

Note that a space after the left bracket and one before the right bracket are essential in this form of the syntax.

7.9.2 Using the case Statement

A multiple test conditional is provided by the **case** command. The basic format of the **case** statement is:


```

case string in
    pattern ) command-list ;;
    ...
    pattern ) command-list ;;
esac

```

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in filename generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the **case** and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a star (*) is the first pattern in a **case**, no other patterns are looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by vertical bars (|).

```

case $i in
    *.c)                cc $i
                        ;;
    *.h | *.sh)         : do nothing
                        ;;
    *)                  echo "$i of unknown type"
                        ;;
esac

```

In the above example, no action is taken for the second set of patterns because the null, colon (:) command is specified. The star (*) is used as a default pattern, because it matches any word.

The exit status of **case** is the exit status of the last command executed in the **case** command. If no commands are executed, then **case** has a zero exit status.

7.9.3 Conditional Looping: **while** and **until**

A **while** command has the general form:

```

while command-list
do
    command-list
done

```

The commands in the first *command-list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second *command-list* are executed. This sequence is repeated as long as the exit status of the first *command-list* is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing **while** with **until**.

Any newline in the above example may be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second *command-list*. If no such command is executed, **while** (or **until**) has a zero exit status.

7.9.4 Looping Over a List: **for**

Often, one wishes to perform some set of operations for each file in a set of files, or execute some command once for each of several arguments. The **for** command can be used to accomplish this. The **for** command has the format:

```

for variable in word-list
do
    command-list
done

```

Here *word-list* is a list of strings separated by blanks. The commands in the *command-list* are executed once for each word in the *word-list*. *Variable* takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following **for** loop causes each of the C source files *xec.c*, *cmd.c*, and *word.c* in the current directory to be compared with a file of the same name in the directory */usr/src/cmd/sh*:

```

for CFILE in xec cmd word
do
    diff ${CFILE}.c /usr/src/cmd/sh/${CFILE}.c
done

```

Note that the first occurrence of CFILE immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the “**in** *word-list*” part of a **for** command; this causes the current set of positional parameters to be used in place of *word-list*. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments. Create a file named *echo2* that contains the following shell script:

```

for word
do echo $word$word
done

```

Give *echo2* execute status:

```

chmod +x echo2

```

Now type the following command:

```

echo2 ma pa bo fi yo no so ta

```

The output from this command is:

```

mama
papa
bobo
fifi
yoyo
nono
soso
tata

```

7.9.5 Loop Control: **break** and **continue**

The **break** command can be used to terminate execution of a **while** or a **for** loop. **Continue** requests the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done**. Exit from *n* levels is obtained by **break** *n*.

The **continue** command causes execution to resume at the nearest enclosing **for**, **while**, or **until** statement, i.e., the one that begins the innermost loop containing the **continue**. You can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:


```

# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do
    echo "Please enter data"
    read response
    case "$response" in
        "done")
            break
            # no more data
            ;;
        "")
            # just a carriage return,
            # keep on going
            continue
            ;;
        *)
            # process the data here
            ;;
    esac
done

```

7.9.6 End-of-File and exit

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a CNTRL-D which is the same as logging out.

The **exit** command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing “exit 0” at the end of the file.

7.9.7 Command Grouping: Parentheses and Braces

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command line—they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you type

```
garble(stuff)
```

the shell prints an error message. Quoted lines, such as

```
garble("stuff")
"garble(stuff)"
```

are interpreted correctly. Other quoting mechanisms are discussed in section 7.2.5, “Quoting Mechanisms”.

This capability of creating a subshell by grouping commands is useful when performing operations without affecting the values of variables in the current shell, or when temporarily changing the working directory and executing commands in the new directory without having to return to the current directory.

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n | lpr& ; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n | lpr &)
```

accomplish the same result: a copy of */usr/docs/otherdir/doc.n* is sent to the lineprinter. (Note that **nroff** is a command available in the XENIX Text Processing System.) However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) may also be used to group commands together. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace may be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

7.9.8 Input/Output Redirection and Control Commands

The shell normally does *not* fork and create a new shell when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if**, **while**, **until**, **case**, and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

1. Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the effect of using parentheses to group commands).
2. Control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

7.9.9 Transfer to Another File and Back: The Dot (.) Command

A command line of the form

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the *.profile* file with:

```
. .profile
```

7.9.10 Interrupt Handling: trap

Shell procedures can use the **trap** command to disable a signal (cause it to be ignored), or redefine its action. The form of the **trap** command is:

```
trap arg signal-list
```

Here *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers as described in *signal(S)* in the XENIX Reference Manual. The most important

of these signals follow:

Number	Signal
00	KILL (CNTRL-U)
01	HANGUP
02	INTERRUPT character
03	QUIT
09	KILL (cannot be caught or ignored)
11	segmentation violation (cannot be caught or ignored)
15	software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in the file *errdirect* when it is interrupted, thus giving the user information as to how much of the job was done:

```
trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
    cd $i
    # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
(trap "echo `pwd` >errdirect" 2 3 15)
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the **trap** command as a signal generated by exiting from a shell. This occurs either with an **exit** command, or by “falling through” to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string ("" or ""), then the signals in the signal list are ignored by the shell.

The **trap** command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
# commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15 (terminate) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotation marks are executed. The **exit** command must be included, or else the shell continues reading commands where it left off when the signal was received. The “trap 0” in the above procedure turns off the original traps 1, 2, 3, and 15 on exits from the shell, so that the **exit** command does not reactivate the execution of the trap commands.

Sometimes the shell continues reading commands after executing trap commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands typed at the terminal until an end-of-file (CNTRL-D) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:


```

d=`pwd`
for i in *
do
    if test -d $d/$i
    then cd $d/$i
        while
            echo "$i:"
            trap exit 2
            read x
        do
            trap : 2
            # ignore interrupts
            eval $x
        done
    fi
done

```

Several **traps** may be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the shell implements the **trap** command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned *after* the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

7.10 Special Shell Commands

There are several special commands that are *internal* to the shell, some of which have already been mentioned. The shell does not fork to execute these commands, so no additional processes are spawned. These commands should be used whenever possible, because they are, in general, faster and more efficient than other XENIX commands. The trade-off for this efficiency is that redirection of input and output is not allowed for most of these special commands.

Several of the special commands have already been described because they affect the flow of control. They are dot (**.**), **break**, **continue**, **exit**, and **trap**. The **set** command is also a special command. Descriptions of the remaining special commands are given here:

:	The null command. This command does nothing and can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#) which can be used to insert comments to the end-of-line. Beware: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.
---	--

cd <i>arg</i>	Make <i>arg</i> the current directory. If <i>arg</i> is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned.
---------------	---

Specifying **cd** with no *arg* is equivalent to typing “cd \$HOME” which takes you to your home directory.

exec arg ... If *arg* is a command, then the shell executes the command without forking and returning to the current shell. This is effectively a “goto” and no new process is created. Input and output redirection arguments are allowed on the command line. If *only* input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.

newgrp arg ... The **newgrp** command is executed, replacing the shell. **Newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any variables that were exported will no longer be marked as such.

read var ... One line (up to a newline) is read from the standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.

readonly var ... The specified variables are made **readonly** so that no subsequent assignments may be made to them. If no arguments are given, a list of all **readonly** and of all exported variables is given.

times The accumulated user and system times for processes run from the current shell are printed.

umask nnn The user file creation mask is set to *nnn*. If *nnn* is omitted, then the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal umask of 137 corresponds to the following bit-mask and permission settings for a newly created file:

User	user	group	other
Octal	1	3	7
bit-mask	001	011	111
permissions	rw-	r--	---

See *umask(C)* in the XENIX *Reference Manual* for information on the value of *nnn*.

wait The shell waits for all currently active child processes to terminate. The exit status of **wait** is always zero.

7.11 Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the *mode* of the file to make it *executable*, thus permitting it to be invoked by

proc args

rather than

sh proc args

The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. To set up a simple procedure, first create a file named *mailall* with the following contents:

```
LETTER=$1
shift
for i in $*
do mail $i <$LETTER
done
```

Next type:

```
chmod +x mailall
```

The new command might then be invoked from within the current directory by typing:

```
mailall letter joe bob
```

Here *letter* is the name of the file containing the message you want to send, and *joe* and *bob* are people you want to send the message to. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If *mailall* were thus created in a directory whose name appears in the user's PATH variable, the user could change working directories and still invoke the *mailall* command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the *dot* command (.) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer writing shell procedures to writing C programs. This is true for several reasons:

1. A shell procedure is easy to create and maintain because it is only a file of ordinary text.
2. A shell procedure has no corresponding object program that must be generated and maintained.
3. A shell procedure is easy to create quickly, use a few times, and then remove.
4. Because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named *bin*. This name is derived from the word "binary", and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their PATH variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it may become difficult to keep track of your environment and efficiency may suffer.

7.12 More About Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

- e This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional

constructs.

- u This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.
- t This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.
- n This is a “don’t execute” flag. On occasion, one may want to check a procedure for syntax errors, but not execute the commands in the procedure. Using “set —nv” at the beginning of a file will accomplish this.
- k This flag causes all arguments of the form *variable=value* to be treated as keyword parameters. When this flag is *not* set, only such arguments that appear before the command name are treated as keyword parameters.

7.13 Supporting Commands and Features

Shell procedures can make use of any XENIX command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use.

7.13.1 Conditional Evaluation: test

The **test** command evaluates the expression specified by its arguments and, if the expression is true, **test** returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. **Test** also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the command list following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets may be used as an alias to **test**, so that

[*expression*]

has the same effect as:

test *expression*

Note that the spaces before and after the *expression* in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

- r *file* True if the named file exists and is readable by the user.
- w *file* True if the named file exists and is writable by the user.
- x *file* True if the named file exists and is executable by the user.
- s *file* True if the named file exists and has a size greater than zero.
- d *file* True if the named file is a directory.
- f *file* True if the named file is an ordinary file.
- z *sl* True if the length of string *sl* is zero.

<code>-n <i>s1</i></code>	True if the length of the string <i>s1</i> is nonzero.
<code>-t <i>fildes</i></code>	True if the open file whose file descriptor number is <i>fildes</i> is associated with a terminal device. If <i>fildes</i> is not specified, file descriptor 1 is used by default.
<code><i>s1</i> = <i>s2</i></code>	True if strings <i>s1</i> and <i>s2</i> are identical.
<code><i>s1</i> != <i>s2</i></code>	True if strings <i>s1</i> and <i>s2</i> are <i>not</i> identical.
<code><i>s1</i></code>	True if <i>s1</i> is <i>not</i> the null string.
<code><i>n1</i> -eq <i>n2</i></code>	True if the integers <i>n1</i> and <i>n2</i> are algebraically equal; other algebraic comparisons are indicated by <code>-ne</code> (not equal), <code>-gt</code> (greater than), <code>-ge</code> (greater than or equal to), <code>-lt</code> (less than), and <code>-le</code> (less than or equal to).

These may be combined with the following operators:

<code>!</code>	Unary negation operator.
<code>-a</code>	Binary logical AND operator.
<code>-o</code>	Binary logical OR operator; it has lower precedence than the logical AND operator (<code>-a</code>).
<code>(<i>expr</i>)</code>	Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to **test**.

7.13.2 Echoing Arguments

The **echo** command has the following syntax:

```
echo [ options ] [ args ]
```

Echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. Often, it is used to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command

```
ls
```

is often replaced by

```
echo *
```

because the latter is faster and prints fewer lines of output.

The `-n` option to **echo** removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow typing on the same line as the prompt:

```
echo -n 'enter name:'
read name
```

The **echo** command also recognizes several escape sequences described in *echo* (C) in the XENIX *Reference Manual*.

7.13.3 Expression Evaluation: **expr**

The **expr** command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; **expr** can be used inside grave accents to set a variable. Some typical examples follow:

```
#          increment $A
A=`expr $a + 1`
#          put third through last characters of
#          $1 into substring
substring=`expr "$1" : '..\(.*\)' ``
#          obtain length of $1
c=`expr "$1" : '.*' ``
```

The most common uses of **expr** are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

7.13.4 True and False

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might type:

```
while true
do echo forever
done
```

This will echo "forever" on the screen until an INTERRUPT is typed.

7.13.5 In-Line Input Documents

Upon seeing a command line of the form

```
command << eofstring
```

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring*. (By appending a minus (–) to the input redirection symbol (<<), leading spaces and tabs are deleted from each line of the input document before the shell passes the line to *command*.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you may quote any character of *eofstring*:

```
command << \eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could type:

```
cat <<– xx
    This message will be printed on the
    terminal with leading tabs and spaces
    removed.

xx
```

This in-line input document feature is most useful in shell procedures. Note that in-line input documents may not appear within grave accents.

7.13.6 Input / Output Redirection Using File Descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the *write(S)* system call (see the *XENIX Reference Manual*). The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By typing

```
fd1 > &fd2
```

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing:

```
command 2> &1
```

If you wanted to redirect both standard output and standard error output to the same file, you would type:

```
command 1>file 2> &1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard input. You could type

```
fda < &fdb
```

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

7.13.7 Conditional Substitution

Normally, the shell replaces occurrences of *\$variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in any one of the following ways:

```
A=
bcd=""
efg=""
set "" ""
```

The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. *Parameter* as used below refers to either a digit or a variable name.

\${variable:-string} If *variable* is set and is nonnull, then substitute the value *\$variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

- `${variable:=string}`** If *variable* is set and is nonnull, then substitute the value *\$variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value *\$variable* in place of this expression. Positional parameters may not be assigned values in this fashion.
- `${variable:?string}`** If *variable* is set and is nonnull, then substitute the value of *variable* for the expression. Otherwise, print a message of the form
- variable: string*
- and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message
- variable: parameter null or not set*
- is printed instead.
- `${variable:+string}`** If *variable* is set and is nonnull, then substitute *string* for this expression. Otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the PATH variable:

`"PATH"=${PATH:-'/bin:/usr/bin'}`

This says, if PATH has ever been set and is not null, then it keeps its current value; otherwise, set it to the string `"/bin:/usr/bin"`.

2. This example automatically assigns the HOME variable a value:

`cd ${HOME:= '/usr/gas'}`

If HOME is set, and is not null, then change directory to it. Otherwise set HOME to the given value and change directory to it.

7.13.8 Invocation Flags

There are five flags that may be specified on the command line when invoking the shell. These flags may not be turned on with the `set` command:

- i** If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.
- s** If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. The shell you get upon logging into the system has the `-s` flag turned on.
- c** When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored. Double quotation marks should be used to enclose a multiword string, in order to allow for variable substitution.
- t** When this flag is on, a single command is read and executed, then the shell exits. This flag is not useful interactively, but is intended for use with C programs.

-r If this flag is present the shell is a restricted shell (see *rsh(C)*).

7.14 Effective and Efficient Shell Programming

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the **time** command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

7.14.1 Number of Processes Generated

When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built in:

break	case	cd	continue	eval
exec	exit	export	for	if
newgrp	read	readonly	set	shift
test	times	trap	umask	until
wait	while	.	:	}

Parentheses, **()**, are built into the shell, but commands enclosed within them are executed as a child process, i.e., the shell does a **fork**, but no **exec**. Any command not in the above list requires both **fork** and **exec**.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

$$\text{processes} = (k * n) + c$$

where *k* and *c* are constants, and *n* may be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero.

Any procedure whose complexity measure includes n^2 terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named *split*, whose text is given below:


```

#      split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
        # read stdin into temp file
        # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
        # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
        # lines with only numbers onto $2
total=`wc -l < temp$$`
end1=`wc -l < $1`
end2=`wc -l < $2`
lost=`expr $total - \($end1 - $start1\) \
- \($end2 - $start2\)`
echo "$total read, $lost thrown away"

```

For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr**. One additional **echo** is executed at the end. If n is the number of lines of input, the number of processes is $2*n + 1$.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C. Shell procedures should not be used to scan or build files a character at a time.

7.14.2 Number of Data Bytes Accessed

It is worthwhile to consider any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to **sort** will be much smaller:

```

sort file | grep pattern
grep pattern file | sort

```

7.14.3 Shortening Directory Searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd**, the **change directory** command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```

ls -l /usr/bin/* > /dev/null
cd /usr/bin; ls -l * > /dev/null

```

The second command will run faster because of the fewer directory searches.

7.14.4 Directory-Search Order and the PATH Variable

The PATH variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current PATH variable. As an example, consider the effect of invoking **nroff** (i.e., */usr/bin/nroff*) when the value of PATH is “*:/bin:/usr/bin*”. The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

This is a total of six directories. A long path list assigned to PATH can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin*. Careless PATH setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

```
:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
/bin::/usr/bin:/usr/john/bin:/usr/localbin
```

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin*.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the PATH variable inside the procedure so that the fewest possible directories are searched in an optimum order.

7.14.5 Good Ways to Set Up Directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 30 files (plus the required *.* and *..*) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a small directory; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 30 or 286 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

7.15 Shell Procedure Examples

The power of the XENIX shell command language is most readily seen by examining how XENIX's many labor-saving utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called “scripts”). Note the use of the number sign (*#*) to introduce comments into shell procedures.

It is intended that the following steps be carried out for each procedure:

1. Place the procedure in a file with the indicated name.
2. Give the file execute permission with the **chmod** command.
3. Move the file to a directory in which commands are kept, such as your own *bin* directory.
4. Make sure that the path of the *bin* directory is specified in the **PATH** variable found in *.profile*.
5. Execute the named command.

BINUNIQ

```
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both */bin* and */usr/bin*. It is done because files in */bin* will “override” those in */usr/bin* during most searches and duplicates need to be weeded out. If the */usr/bin* file is obsolete, then space is being wasted; if the */bin* file is outdated by a corresponding entry in */usr/bin* then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of “sort|uniq” to find matches and duplications.

COPYPAIRS

```
#      Usage: ctypairs file1 file2 ...
#      Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
    cp $1 $2
    shift; shift
done
if test "$1" != ""
    then echo "$0: odd number of arguments"
fi
```

This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop, because you can adjust the positional parameters with the **shift** command to handle related arguments.

COPYTO

```

#      Usage: copyto dir file ...
#      Copies argument files to "dir",
#      making sure that at least
#      two arguments exist, that "dir" is a directory,
#      and that each additional argument
#      is a readable file.
if test $# -lt 2
then    echo "$0: usage: copyto directory file ..."
elif test ! -d $1
then    echo "$0: $1 is not a directory";
else    dir=$1; shift
        for eachfile
        do        cp $eachfile $dir
done
fi

```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first; the original **\$1** is shifted off.

DISTINCT1

```

#      Usage: distinct1
#      Reads standard input and reports list of
#      alphanumeric strings that differ only in case,
#      giving lowercase form of each.
tr -cs 'A-Za-z0-9' '\012' | sort -u | \
tr 'A-Z' 'a-z' | sort | uniq -d

```

This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It may not be immediately obvious how this command works. You may wish to consult *tr(C)*, *sort(C)*, and *uniq(C)* in the *XENIX Reference Manual* if you are completely unfamiliar with these commands. The **tr** command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The “**uniq -d**” prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. The first line below is equivalent to the last two lines, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
```

```

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]

```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to

create a series of transformations that will convert the input to the desired output.

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

DRAFT

```
#      Usage: draft file(s)
#      Print manual pages for Diablo printer.
for i in $*
do nroff -man $i | lpr
done
```

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
#      Usage: edfind file arg
#      Finds the last occurrence in "file" of a line
#      whose beginning matches "arg", then prints
#      3 lines (the one before, the line itself,
#      and the one after)
ed - $1 <<-EOF
    ?^$2?
    -,+p
    q
EOF
```

This illustrates the practice of using **ed** in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
#      Usage: edlast file
#      Prints the last line of file,
#      then deletes that line.
ed - $1 <<-\!
    $p
    $d
    w
    q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation point (!). Variable substitution is prohibited within the input text because of the backslash.

FSPLIT

```

#      Usage: fsplit file1 file2
#      Reads standard input and divides it into 3 parts
#      by appending any line containing at least one letter
#      to file1, appending any line containing digits but
#      no letters to file2, and by throwing the rest away.
count=0 gone=0
while read next
do
    count=`expr $count + 1 `
    case "$next" in
        *[A-Za-z]*)
            echo "$next" >> $1 ;;
        *[0-9]*)
            echo "$next" >> $2 ;;
        *)
            gone=`expr $gone + 1 `
    esac
done
echo "$count lines read, $gone thrown away"

```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file. Note the use of the **expr** command.

Don't use the shell to read a line at a time unless you must—it can be an extremely slow process.

LISTFIELDS

```
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a colon on a newline. Thus, if given the following input

```
joe newman: 13509 NE 78th St: Redmond, Wa 98062
```

listfields will produce this:

```
joe newman
13509 NE 78th St
Redmond, Wa 98062
```

Note the use of the **tr** command to transpose colons to linefeeds.

MKFILES

```
#      Usage: mkfiles pref [quantity]
#      Makes "quantity" files, named pref1, pref2, ...
#      Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
    > $1$i
    i=`expr $i + 1`
done
```

The *mkfiles* procedure uses output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop.

NULL

```
#      Usage: null files
#      Create each of the named files as an empty file.
for eachfile
do
    > $eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
#      Usage: phone initials ...
#      Prints the phone numbers of the
#      people with the given initials.
echo 'inits      ext      home'
grep "$1" << END
    jfk      1234      999-2345
    lbj      2234      583-2245
    hst      3342      988-1010
    jqa      4567      555-1234
END
```

This procedure is an example of using an in-line input script to maintain a small data base.

TEXTFILE

```

if test "$1" = "-s"
then
#      Return condition code
      shift
      if test -z "$0 $*" # check return value
      then
          exit 1
      else
          exit 0
      fi
fi

if test $# -lt 1
then  echo "$0: Usage: $0 [ -s ] file ..." 1> &2
      exit 0
fi

file $* | fgrep ' text' | sed 's/:          .*/'

```

To determine which files in a directory contain only textual information, *textfile* filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

```
pr textfile *` | lpr
```

This procedure also uses an `-s` flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```

#      Usage: writemail message user
#      If user is logged in,
#      writes message to terminal;
#      otherwise, mails it to user.
echo "$1" | { write "$2" | mail "$2" ;}

```

This procedure illustrates the use of command grouping. The message specified by `$1` is piped to both the **write** command and, if **write** fails, to the **mail** command.

7.16 Shell Grammar

item: *word*
 input-output
 name = value

simple-command: *item*
 simple-command item

command: *simple-command*
 (command-list)
 { command-list }
 for *name* **do** *command-list* **done**
 for *name* **in** *word* **do** *command-list* **done**
 while *command-list* **do** *command-list* **done**
 until *command-list* **do** *command-list* **done**
 case *word* **in** *case-part* **esac**
 if *command-list* **then** *command-list* **else-part** **fi**

pipeline: *command*
 pipeline | command

andor: *pipeline*
 andor & & pipeline
 andor | pipeline

command-list: *andor*
 command-list ;
 command-list &
 command-list ; andor
 command-list & andor

input-output: *> file*
 < file
 << word
 >> word

file: *word*
 & digit
 & -

case-part: *pattern*) *command-list* ;;

pattern: *word*
 pattern | *word*

else-part: **elif** *command-list* **then** *command-list* *else-part*
 else *command-list*
 empty

empty:

word: *a sequence of nonblank characters*

name: *a sequence of letters, digits, or underscores*
 starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

Metacharacters and Reserved Words

a. Syntactic

	Pipe symbol
& &	And-if symbol
	Or-if symbol
;	Command separator
::	Case delimiter
&	Background commands
()	Command grouping
<	Input redirection
< <	Input from a here document
>	Output creation
<	Output append
#	Comment to end of line

b. Patterns

*	Match any character(s) including none
?	Match any single character
[...]	Match any of enclosed characters

c. Substitution

\${...}	Substitute shell variable
`...`	Substitute command output

d. Quoting

\	Quote next character as literal with no special meaning
'...'	Quote enclosed characters excepting the back quotation marks (`)
"..."	Quote enclosed characters excepting: \$ ` \"

e. Reserved words

if	esac
then	for
else	while
elif	until
fi	do
case	done
in	{ }

Chapter 8

BC: A Calculator

8.1 Introduction	1
8.2 Demonstration	1
8.3 Tasks	3
8.3.1 Computing with Integers	3
8.3.2 Specifying Input and Output Bases	4
8.3.3 Scaling Quantities	4
8.3.4 Using Functions	5
8.3.5 Using Subscripted Variables	6
8.3.6 Using Control Statements: if, while and for	7
8.3.7 Using Other Language Features	8
8.4 Language Reference	10
8.4.1 Tokens	10
8.4.2 Expressions	10
8.4.3 Function Calls	11
8.4.4 Unary Operators	11
8.4.5 Multiplicative Operators	12
8.4.6 Additive Operators	12
8.4.7 Assignment Operators	12
8.4.8 Relational Operators	13
8.4.9 Storage Classes	13
8.4.10 Statements	13

8.1 Introduction

BC is a program that can be used as an arbitrary precision arithmetic calculator. BC's output is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers. Although you can write substantial programs with BC, it is often used as an interactive tool for performing calculator-like computations. The language supports a complete set of control structures and functions that can be defined and saved for later execution. The syntax of BC has been deliberately selected to agree with the C language; those who are familiar with C will find few surprises. A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Common uses for BC are:

- Computation with large integers.
- Computations accurate to many decimal places.
- Conversions of numbers from one base to another base.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal simply by setting the output base equal to 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine, so manipulation of numbers with many hundreds of digits is possible.

8.2 Demonstration

This demonstration is designed to show you:

- How to get into and out of BC.
- How to perform simple computations.
- How expressions are formed and evaluated.
- How to assign values to registers.

A normal session with BC begins by invoking the program with the command:

```
bc
```

To exit BC type

```
quit
```

or press CNTRL-D. Once you have entered BC, you can use it very much like a normal calculator. As with the XENIX shell, commands are read as command-lines, so each line that you type must be terminated by a RETURN. Throughout this chapter, the RETURN is implied at the end of each command line. Within BC, normal processing of other keys, such as BKSP and INTERRUPT, also works.

For example, type the simple integer 5:

```
5
```

Output is immediately echoed on the next line to the standard output, which is normally the terminal screen:

5

Here "5" is a simple numeric expression. However, if you type the expression

5*5.25

(where the star (*) is the multiplication operator) a computation is executed and the result printed on the next line:

26.25

What has happened here is that the line "5*5.25" has been evaluated, i.e., the expression has been reduced to its most elementary form, which is the number 26.25. The process of evaluation normally involves some type of computation such as multiplication, division, addition, or subtraction. For example, all four of these operations are involved in the following expression:

(10*5)+50-(50/2)

When this expression is evaluated, the subexpressions within parentheses are evaluated first, just as they would be with simple algebra, so that an intermediate step in the evaluation is "50+50-25" which ultimately reduces to the number "75".

The simple addition

10.45+5.5555555

produces the output:

16.0055555

Note how precision is retained in the above result.

The two-part multiplication

(8*9)*7

produces the answer:

504

The last part of this demonstration shows you how to store values in special alphabetic registers. For example, type:

a=100 ; b=5

What happens here is that the registers "a" and "b" are assigned the values 100 and 5, respectively. The semicolon is used here to place multiple BC statements on a single line, just as it is used in the XENIX shell. This command line produces no output because assignment statements are not considered expressions. However, the registers "a" and "b" can now be used in expressions. Thus you can now type

a*b; a+b

to produce:

500

105

To exit BC, remember to type

quit

or press CNTRL-D.

This ends the demonstration. Following sections describe use of BC in more detail. The final section of this chapter is a BC language reference.

8.3 Tasks

This section describes how to perform common BC tasks. Mastery of these tasks should turn you into a competent BC user.

8.3.1 Computing with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type

```
142857 + 285714
```

and press RETURN, BC responds immediately with the line:

```
428571
```

Other operators also can be used. The complete list includes:

```
+ - * / % ^
```

They indicate addition, subtraction, multiplication, division, modulo (remaindering), and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error message.

Any term in an expression can be prefixed with a minus sign to indicate that it is to be negated (this is the “unary” minus sign). For example, the expression

```
7+-3
```

is interpreted to mean that -3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in FORTRAN, with exponentiation (^) performed first, then multiplication (*), division (/), modulo (%), and finally, addition (+), and subtraction (-). The contents of parentheses are evaluated before expressions outside the parentheses. All of the above operations are performed from left to right, except exponentiation, which is performed from right to left. Thus the following two expressions

```
a^b^c and a^(b^c)
```

are equivalent, as are the two expressions:

```
a*b*c and (a*b)*c
```

BC shares with FORTRAN and C the convention that $a/b*c$ is equivalent to $(a/b)*c$.

Internal storage registers to hold numbers have single lowercase letter names. The value of an expression can be assigned to a register in the usual way, thus the statement

```
x = x + 3
```

has the effect of increasing by 3 the value of the contents of the register named “x”. When, as in this case, the outermost operator is the assignment operator (=), then the assignment is performed but the result is not printed. There are 26 available named storage registers, one for each letter of the alphabet.

There is also a built-in square root function whose result is truncated to an integer (See also Section 8.5, “Scaling”). For example, the lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

8.3.2 Specifying Input and Output Bases

There are special internal quantities in BC, called *ibase* and *obase*. *Ibase* is initially set to 10, and determines the base used for interpreting numbers that are read by BC. For example, the lines

```
ibase = 8
11
```

produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. However, beware of trying to change the input base back to decimal by typing:

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement has no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15, respectively. These characters *must* be uppercase and not lowercase. The statement

```
ibase = A
```

changes you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted; however no mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

Obase is used as the base for output numbers. The value of *obase* is initially set to a decimal 10. The lines

```
obase = 16
1000
```

produce the output line:

```
3E8
```

This is interpreted as a three-digit hexadecimal number. Very large output bases are permitted. For example, large numbers can be output in groups of five digits by setting *obase* to 100000. Even strange output bases, such as negative bases, and 1 and 0, are handled correctly.

Very large numbers are split across lines with seventy characters per line. A split line that continues on the next line ends with a backslash (\). Decimal output conversion is fast, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow.

Remember that *ibase* and *obase* do not affect the course of internal computation or the evaluation of expressions; they only affect input and output conversion.

8.3.3 Scaling Quantities

A special internal quantity called *scale* is used to determine the scale of calculated quantities. Numbers can have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

Addition, subtraction

The scale of the result is the larger of the scales of the two operands. There is never any truncation of the result.

Multiplication	The scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands, and subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity, <i>scale</i> .
Division	The scale of a quotient is the contents of the internal quantity, <i>scale</i> .
Modulo	The scale of a remainder is the sum of the scales of the quotient and the divisor.
Exponentiation	The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.
Square Root	The scale of a square root is set to the maximum of the scale of the argument and the contents of <i>scale</i> .

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded truncation is performed without rounding.

The contents of *scale* must be no greater than 99 and no less than 0. It is initially set to 0.

The internal quantities *scale*, *ibase*, and *base* can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of *scale* by one, and the line

```
scale
```

causes the current value of *scale* to be printed.

The value of *scale* retains its meaning as a number of decimal digits to be retained in internal computation even when *ibase* or *obase* are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

8.3.4 Using Functions

The name of a function is a single lowercase letter. Function names are permitted to use the same letters as simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace (`}`). Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms:

```
return
return(x)
```

In the first case, the returned value of the function is 0; in the second, it is the value of the expression in parentheses.

Variables used in functions can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one **auto** statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the

function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions can be called recursively and the automatic variables at each call level are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function, with the single exception that they are given a value on entry to the function. An example of a function definition follows:

```
define a(x,y){
    auto z
    z = x*y
    return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name, followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

If the function "a" is defined as shown above, then the line

```
a(7,3.14)
```

would print the result:

```
21.98
```

Similarly, the line

```
x = a(a(3,4),5)
```

would cause the value of "x" to become 60.

Functions can require no arguments, but still perform some useful operation or return a useful result. Such functions are defined and called using parentheses with nothing between them. For example:

```
b ()
```

calls the function named *b*.

8.3.5 Using Subscripted Variables

A single lowercase letter variable name followed by an expression in brackets is called a subscripted variable and indicates an array element. The variable name is the name of the array and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted in BC. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables can be freely used in expressions, in function calls and in return statements.

An array name can be used as an argument to a function, as in:

```
f(a[ ])
```

Array names can also be declared as automatic in a function definition with the use of empty brackets:

```
define f(a[ ])
    auto a[ ]
```

When an array name is so used, the entire contents of the array are copied for the use of the function, then thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other context.

8.3.6 Using Control Statements: if, while and for

The **if**, **while**, and **for** statements are used to alter the flow within programs or to cause iteration. The range of each of these statements is a following statement or compound statement consisting of a collection of statements enclosed in braces. They are written as follows:

```

if(relation) statement
while(relation) statement
for(expression1 ; relation ; expression2) statement

if(relation) {statements}
while(relation) {statements}
for(expression1 ; relation ; expression2) {statements}

```

A relation in one of the control statements is an expression of the form

expression1 rel-op expression2

where the two expressions are related by one of the six relational operators:

< > <= >= == !=

Note that a double equal sign (==) stands for “equal to” and an exclamation-equal sign (!=) stands for “not equal to”. The meaning of the remaining relational operators is their normal arithmetic and logical meaning.

Beware of using a single equal sign (=) instead of the double equal sign (==) in a relational. Both of these symbols are legal, so you will not get a diagnostic message. However, the operation will not perform the intended comparison.

The **if** statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in the sequence.

The **while** statement causes repeated execution of its range as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the **while** statement.

The **for** statement begins by executing *expression1*. Then the relation is tested and, if true, the statements in the range of the **for** statement are executed. Then *expression2* is executed. The relation is tested, and so on. The typical use of the **for** statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10.

The following are some examples of the use of the control statements:

```

define f(n){
    auto i, x
    x=1
    for(i=1; i<=n; i=i+1) x=x*i
    return(x)
}

```

The line

```
f(a)
```

prints “a” factorial if “a” is a positive integer.

The following is the definition of a function that computes values of the binomial coefficient (“m” and “n” are assumed to be positive integers):

```
define b(n,m){
    auto x, j
    x=1
    for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
    return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard to possible truncation errors:

```
scale = 20
define e(x){
    auto a, b, c, d, n
    a = 1
    b = 1
    c = 1
    d = 0
    n = 1
    while(1==1){
        a = a*x
        b = b*n
        c = c + a/b
        n = n + 1
        if(c==d) return(c)
        d = c
    }
}
```

8.3.7 Using Other Language Features

Some language features that every user should know about are listed below.

- Normally, statements are typed one to a line. It is also permissible to type several statements on a line if they are separated by semicolons.
- If an assignment statement is placed in parentheses, it then has a value and can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

The following is an example of a use of the value of an assignment statement even when it is not placed in parentheses:

```
x = a[i=i+1]
```

This causes a value to be assigned to “x” and also increments “i” before it is used as a

subscript.

- The following constructions work in BC in exactly the same manner as they do in the C language:

Construction	Equivalent
<code>x=y=z</code>	<code>x =(y=z)</code>
<code>x =+ y</code>	<code>x = x+y</code>
<code>x =- y</code>	<code>x = x-y</code>
<code>x =* y</code>	<code>x = x*y</code>
<code>x =/ y</code>	<code>x = x/y</code>
<code>x =% y</code>	<code>x = x%y</code>
<code>x ^= y</code>	<code>x = x^y</code>
<code>x++</code>	<code>(x=x+1)-1</code>
<code>x--</code>	<code>(x=x-1)+1</code>
<code>++x</code>	<code>x = x+1</code>
<code>--x</code>	<code>x = x-1</code>

Even if you don't intend to use these constructions, if you type one inadvertently, something legal but unexpected may happen. Be aware that in some of these constructions spaces are significant. There is a real difference between "`x=-y`" and "`x= -y`". The first replaces "`x`" by "`x-y`" and the second by "`-y`".

- The comment convention is identical to the C comment convention. Comments begin with "`/*`" and end with "`*/`".
- There is a library of math functions that can be obtained by typing

`bc -l`

when you invoke BC. This command loads the library functions sine, cosine, arctangent, natural logarithm, exponential, and Bessel functions of integer order. These are named "`s`", "`c`", "`a`", "`l`", "`e`", and "`j(n,x)`", respectively. This library sets *scale* to 20 by default.

- If you type

`bc file ...`

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you can load your own programs and function definitions.

8.4 Language Reference

This section is a comprehensive reference to the BC language. It contains a more concise description of the features mentioned in earlier sections.

8.4.1 Tokens

Tokens are keywords, identifiers, constants, operators, and separators. Token separators can be blanks, tabs or comments. Newline characters or semicolons separate statements.

Comments	Comments are introduced by the characters “/*” and are terminated by “*/”.														
Identifiers	There are three kinds of identifiers: ordinary identifiers, array identifiers and function identifiers. All three types consist of single lowercase letters. Array identifiers are followed by square brackets, enclosing an optional expression describing a subscript. Arrays are singly dimensioned and can contain up to 2048 elements. Indexing begins at 0 so an array can be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, enclosing optional arguments. The three types of identifiers do not conflict; a program can have a variable named “x”, an array named “x”, and a function named “x”, all of which are separate and distinct.														
Keywords	<p>The following are reserved keywords:</p> <table> <tr><td>ibase</td><td>if</td></tr> <tr><td>obase</td><td>break</td></tr> <tr><td>scale</td><td>define</td></tr> <tr><td>sqrt</td><td>auto</td></tr> <tr><td>length</td><td>return</td></tr> <tr><td>while</td><td>quit</td></tr> <tr><td>for</td><td></td></tr> </table>	ibase	if	obase	break	scale	define	sqrt	auto	length	return	while	quit	for	
ibase	if														
obase	break														
scale	define														
sqrt	auto														
length	return														
while	quit														
for															
Constants	Constants are arbitrarily long numbers with an optional decimal point. The hexadecimal digits A-F are also recognized as digits with decimal values 10–15, respectively.														

8.4.2 Expressions

All expressions can be evaluated to a value. The value of an expression is always printed unless the main operator is an assignment. The precedence of expressions (i.e., the order in which they are evaluated) is as follows:

Function calls

Unary operators

Multiplicative operators

Additive operators

Assignment operators

Relational operators

There are several types of expressions:

Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

identifiers

Simple identifiers are named expressions. They have an initial value of zero.

array-name[expression]

Array elements are named expressions. They have an initial value of zero.

scale, ibase and obase

The internal registers *scale*, *ibase*, and *obase* are all named expressions. *Scale* is the number of digits after the decimal point to be retained in arithmetic operations and has an initial value of zero. *Ibase* and *obase* are the input and output number radices respectively. Both *ibase* and *obase* have initial values of 10.

Constants

Constants are primitive expressions that evaluate to themselves.

Parenthetic Expressions

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter normal operator precedence.

Function Calls

Function calls are expressions that return values. They are discussed in section 8.10.3.

8.4.3 Function Calls

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. The syntax is as follows:

function-name ([*expression* [, *expression* ...]])

A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement, or 0 if no expression is provided or if there is no return statement. Three built-in functions are listed below:

sqrt(*expr*) The result is the square root of the expression and is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of *scale*, whichever is larger.

length(*expr*) The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale(*expr*) The result is the scale of the expression. The scale of the result is zero.

8.4.4 Unary Operators

The unary operators bind right to left.

-*expr* The result is the negative of the expression.

- ++ named_expr*** The named expression is incremented by one. The result is the value of the named expression after incrementing.
- named_expr*** The named expression is decremented by one. The result is the value of the named expression after decrementing.
- named_expr++*** The named expression is incremented by one. The result is the value of the named expression before incrementing.
- named_expr--*** The named expression is decremented by one. The result is the value of the named expression before decrementing.

8.4.5 Multiplicative Operators

The multiplicative operators (*, /, and %) bind from left to right.

- expr*expr*** The result is the product of the two expressions. If “a” and “b” are the scales of the two expressions, then the scale of the result is:
- $$\min(a+b, \max(\text{scale}, a, b))$$
- expr/expr*** The result is the quotient of the two expressions. The scale of the result is the value of *scale*.
- expr%expr*** The modulo operator (%) produces the remainder of the division of the two expressions. More precisely, $a\%b$ is $a-a/b*b$. The scale of the result is the sum of the scale of the divisor and the value of *scale*.
- expr^expr*** The exponentiation operator binds right to left. The result is the first expression raised to the power of the second expression. The second expression must be an integer. If “a” is the scale of the left expression and “b” is the absolute value of the right expression, then the scale of the result is:
- $$\min(a*b, \max(\text{scale}, a))$$

8.4.6 Additive Operators

The additive operators bind left to right.

- expr+expr*** The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.
- expr-expr*** The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

8.4.7 Assignment Operators

The assignment operators listed below assign values to the named expression on the left side.

- named_expr=expr***
This expression results in assigning the value of the expression on the right to the named expression on the left.
- named_expr+=expr***
The result of this expression is equivalent to $\text{named_expr}=\text{named_expr}+\text{expr}$.

named_expr = -expr

The result of this expression is equivalent to *named_expr = named_expr - expr*.

*named_expr = *expr*

The result of this expression is equivalent to *named_expr = named_expr * expr*.

named_expr = /expr

The result of this expression is equivalent to *named_expr = named_expr / expr*.

named_expr = %expr

The result of this expression is equivalent to *named_expr = named_expr % expr*.

named_expr = ^expr

The result of this expression is equivalent to *named_expr = named_expr ^ expr*.

8.4.8 Relational Operators

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement, or inside a **for** statement. These operators are listed below:

expr < expr

expr > expr

expr <= expr

expr >= expr

expr == expr

expr != expr

8.4.9 Storage Classes

There are only two storage classes in BC: global and automatic (local). Only identifiers that are to be local to a function need to be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions.

All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They, therefore, do not retain values between function calls. Note that **auto** arrays are specified by the array name, followed by empty square brackets.

Automatic variables in BC do not work the same way as in C. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

8.4.10 Statements

Statements must be separated by a semicolon or a newline. Except where altered by control statements, execution is sequential. There are four types of statements: expression statements, compound statements, quoted string statements, and built-in statements. Each kind of statement is discussed below:

Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements

Statements can be grouped together and used when one statement is expected by surrounding them with curly braces ({ and }).

Quoted string statements

For example

"string"

prints the string inside the quotation marks.

Built-in statements

Built-in statements include **auto**, **break**, **define**, **for**, **if**, **quit**, **return**, and **while**.

The syntax for each built-in statement is given below:

Auto statement

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition. Syntax of the auto statement is:

auto identifier [, identifier]

Break statement

The **break** statement causes termination of a **for** or **while** statement. Syntax for the break statement is:

break

Define statement

The **define** statement defines a function; parameters to the function can be ordinary identifiers or array names. Array names must be followed by empty square brackets. The syntax of the define statement is:

define ([parameter [, parameter ...]]) {statements}

For statement

The **for** statement is the same as:

```
first-expression
while(relation) {
    statement
    last-expression
}
```

All three expressions must be present. Syntax of the for statement is:

for (expression; relation; expression) statement

If statement

The statement is executed if the relation is true. The syntax is as follows:

if (relation) statement

Quit statement

The **quit** statement stops execution of a BC program and returns control to XENIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement. Note that entering a CNTRL-D at the keyboard is the same as typing "quit". The syntax of the quit statement is as follows:

quit

Return statement

The **return** statement terminates a function, pops its auto variables off the stack, and specifies the result of the function. The result of the function is the result of the expression in parentheses. The first form is equivalent to "return(0)". The syntax of the return statement is as follows:

return(expr)

While statement

The statement is executed while the relation is true. The test occurs before each execution of the statement. The syntax of the while statement is as follows:

while (relation) statement

Chapter 9

Building a Uucp System

9.1	Introduction	1
9.2	Uucp – System to System File Copy	1
9.2.1	Copying Files to a Local Destination	2
9.2.2	Receiving Files from Other Systems	2
9.2.3	Sending Files to Remote Systems	3
9.2.4	Copying Files Between Systems	3
9.3	Uux – System To System Execution	3
9.4	Uucico – Copy In, Copy Out	5
9.4.1	Scanning For Work	5
9.4.2	Calling a Remote System	6
9.4.3	Selecting Line Protocol	6
9.4.4	Processing Work	7
9.4.5	Terminating a Conversation	7
9.5	Uuxqt – Uucp Command Execution	7
9.6	Uulog – Uucp Log Inquiry	8
9.7	Uuclean – Uucp Spool Directory Cleanup	8
9.8	Security	8
9.9	Installing a Uucp System	9
9.9.1	Modifying the <i>/etc/systemid</i> File	9
9.9.2	Creating the Required Files	9
9.10	Maintaining the System	12
9.10.1	SEQF – sequence check file	12
9.10.2	TM – temporary data files	12
9.10.3	LOG – log entry files	13
9.10.4	STST – system status files	13
9.10.5	LCK – lock files	13
9.10.6	Creating Shell Files	13
9.10.7	Defining Login Entries	14
9.10.8	Setting File Modes	14

9.1 Introduction

The uucp system is a series of programs designed to permit communication between XENIX systems using dial-up communication lines. Uucp provides file transfer and remote command execution through a batch-type operation. Files are created in a spool directory for processing by the uucp daemons. There are three types of files used for the execution of work:

Data files	Contain data for transfer to remote systems
Work files	Contain directions for file transfers between systems
Execution files	Contain directions for XENIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp	This program creates work and gathers data files in the spool directory for the transmission of files.
uux	This program creates work files, execute files and gathers data files for the remote execution of XENIX commands.
uucico	This program executes the work files for data transmission.
uuxqt	This program executes the execution files for XENIX command execution.

The secondary programs are:

uulog	This program updates the log file with new entries and reports on the status of uucp requests.
uuclean	This program removes old files from the spool directory.

This chapter describes the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

For hardwired communications between XENIX systems, use the Micnet network described in the *XENIX Operations Guide*.

9.2 Uucp – System to System File Copy

The *uucp* program is the user's primary interface with the system. The *uucp* program was designed to look like the *cp* command. The syntax is

uucp [*option*] ... *source* ... *destination*

where *source* and *destination* may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

-d	Make directories when necessary for copying the file.
-c	Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

-gletter Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

-m Send mail on completion of the work.

The following options are used primarily for debugging:

-r Queue the job but do not start *uucico* program.

-sdir Use directory *dir* for the spool directory.

-xnum Use *num* as the level of debugging output.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "**[]*". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

```
uucp *.c usg!/usr/dan
```

will set up the transfer of all files whose names end with *.c* to the */usr/dan* directory on the *usg* machine.

The source and/or destination names may also contain a *~user* prefix. This translates to the login directory on the specified system. For names with partial pathnames, the current directory is prepended to the file name. File names with "*../*" are not permitted.

The command

```
uucp usg!~dan/*.h ~dan
```

will set up the transfer of files whose names end with *.h* in dan's login directory on system *usg* to dan's local login directory.

For each source file, the program will check the source and destination filenames and the system-part of each to classify the work into one of five types:

1. Copy source to destination on local system.
2. Receive files from other systems.
3. Send files to a remote systems.
4. Send files from remote systems to another remote system.
5. Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the **-r** option was specified).

9.2.1 Copying Files to a Local Destination

A **cp** command is used to do type 1 work. The **-d** and the **-m** options are not honored in this case.

9.2.2 Receiving Files from Other Systems

For type 2 work, a one line work file is created for each file requested and put in the spool directory

with the following fields, each separated by a blank.

- [1] R
- [2] The full pathname of the source or a `~user/pathname`. The `~user` part will be expanded on the remote system.
- [3] The full pathname of the destination file. If the `~user` notation is used, it will be immediately expanded to be the login directory for the user.
- [4] The user's login name.
- [5] A “—” followed by an option list. (Only the `—m` and `—d` options will appear in this list.)

9.2.3 Sending Files to Remote Systems

For type 3 work, a work file is created for each source file and the source file is copied into a data file in the spool directory. (A `—c` option on the `uucp` program will prevent the data file from being made. In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

- [1] S
- [2] The full pathname of the source file.
- [3] The full pathname of the destination or `~user/filename`.
- [4] The user's login name.
- [5] A “—” followed by an option list.
- [6] The name of the data file in the spool directory.
- [7] The file mode bits of the source file in octal print format (e.g. 0666).

9.2.4 Copying Files Between Systems

For type 4 and 5 work, `uucp` generates a `uucp` command line and sends it to the remote machine; the remote `uucico` executes the command line.

9.3 Uux — System To System Execution

The `uux` command is used to set up the execution of a XENIX command where the execution machine and/or some of the files are remote. The syntax of the `uux` command is

```
uux [ — ] [ option ] ... command-string
```

where *command-string* is made up of one or more arguments. All special shell characters such as “<>|” must be quoted either by quoting the entire command string or quoting the character as a separate argument. Within the command string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a “!” will not be treated as files. (They will not be copied to the execution machine.) The `—` option is used to indicate that the standard input for the given command should be inherited from the standard input of the `uux` command. The options, essentially for debugging, are:

- r** Do not start *uucico* or *uuxqt* after queuing the job
- xnum** Use *num* as the level of debugging output.

The command

```
pr abc | uux - usg!lpr
```

will set up the output of "pr abc" as standard input to an lpr command to be executed on system *usg*.

Uux generates an execute file which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The execute file will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

```
U user system
```

where the *user* and *system* are the requester's login name and system.

Required File Line

```
F filename real-name
```

where the *filename* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the execute file. The *uuxqt* program will check for the existence of all required files before the command is executed.

Standard Input Line

```
I filename
```

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the - option is used. If a standard input is not specified, */dev/null* is used.

Standard Output Line

```
O filename system-name
```

The standard output is specified by a ">" within the command-string. If a standard output is not specified, */dev/null* is used. (Note that the use of ">>" is not implemented.)

Command Line

```
C command [ arguments ] ...
```

The *arguments* are those specified in the command string. The standard input and standard output will not appear on this line. All required files will be moved to the execution directory (a subdirectory of the spool directory) and the XENIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell PATH statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

9.4 Uucico – Copy In, Copy Out

The *uucico* program will perform the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started by a system daemon, by one of the *uucp*, *uux*, *uuxqt*, or *uucico* programs, by the user (this is usually for testing), or by a remote system. (The *uucico* program should be specified as the shell field in the */etc/passwd* file for the *uucp* logins.)

When started by method a daemon, a program, or the user, the program is considered to be in MASTER mode. In this mode, a connection will be made to a remote system. If started by a remote system, the program is considered to be in SLAVE mode.

The MASTER mode will operate in one of two ways. If no system name is specified (the *-s* option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

- rl* Start the program in MASTER mode. This is used when *uucico* is started by a program or *cron* shell.
- ssys* Do work only for system *sys*. If *-s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.
- Ssys* Similar to *-s* except *uucio* ignores the times given in the *L.sys* file.

The following operations are used primarily for debugging:

- ddir* Use directory *dir* for the spool directory.
- xnum* Use *num* as the level of debugging output.

The next part of this section will describe the major steps within the *uucico* program.

9.4.1 Scanning For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where *type* may be “C” for copy command file, “D” for data file, “X” for execute file, *system-name* is the remote system, *grade* is a character, and *number* is a four digit, padded sequence number.

The file

C.res45n0031

is a work file for a file transfer between the local machine and the *res45* machine.

The scan for work is done by looking through the spool directory for work files (files with prefix C.). A list is made of all systems to be called. *Uucico* will then call each system and process all work files.

9.4.2 Calling a Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems. The lock filename has the form

LCK..str

The system name is found in the *L.sys* file. The information contained for each system is;

- [1] System name
- [2] Times to call the system (days-of-week and times-of-day)
- [3] Device or device type to be used for call
- [4] line speed
- [5] phone number if field [3] is "ACU" or the device name (same as field [3]) if not
- [6] Login information (multiple fields)

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using device type and line speed fields from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy these fields until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the login information in the last field of *L.sys* is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the SLAVE system. The SLAVE sends a message to let the MASTER know it is ready to receive the system identification and conversation sequence number. The response from the MASTER is verified by the SLAVE and if acceptable, protocol selection begins. The SLAVE can also reply with a call-back required message in which case, the current conversation is terminated.

9.4.3 Selecting Line Protocol

The remote system sends a message

Pproto-list

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks the protocol list for a letter corresponding to an available line protocol and returns a use protocol message. The message has the form

Ucode

where *code* is either a one character protocol letter or “N” which means there is no common protocol.

9.4.4 Processing Work

The initial role of MASTER or SLAVE for the work processing is the mode in which each program starts. (The MASTER has been specified by the `-r1` option.) The MASTER program does a work search similar to the one used in the section “Scanning For Work” above.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S	Send a file
R	Receive a file
C	Copy complete
X	Execute a <i>uucp</i> command
H	Hangup

The MASTER will send *R*, *S*, or *X* messages until all work from the spool directory is complete, at which point an *H* message is sent. The SLAVE will reply with the first letter of the request and either the letter “Y” or “N” for yes or no. For example, the message “SY” indicates that it is okay to send a file.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message “CY” will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a “CN” message is sent. (In the case of “CN”, the transferred file will be in the spool directory with a name beginning with “TM”.) The requests and results are logged on both systems.

The hangup response is determined by the SLAVE program by a work scan of the spool directory. If work for the remote system exists in the SLAVE’s spool directory, an “HN” message is sent and the programs switch roles. If no work exists, an “HY” response is sent.

9.4.5 Terminating a Conversation

When a “HY” message is received by the MASTER it is echoed back to the SLAVE and the protocols are turned off. Each program sends a final “OO” message to the other. The original SLAVE program will clean up and terminate. The MASTER will proceed to call other systems and process work as long as possible or terminate if a `-s` option was specified.

9.5 Uuxqt — Uucp Command Execution

The *uuxqt* program is used to process execute files generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for execute files (prefix *X*). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The execute file is described in the section “Uux - System to System Copy” above.

The execution is accomplished by executing the shell command

```
sh -c
```

with the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

9.6 Uulog – Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to append these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

-ssys Print entries where *sys* is the remote system name

-uuser Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

9.7 Uuclean – Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than three days old. These are usually files for work which can not be completed.

The options available are:

-ddir The directory to be scanned is *dir*.

-m Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)

-nhours Change the aging time from 72 hours to *hours* hours.

-ppre Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)

-xnum Use *num* as the level of debugging output desired.

9.8 Security

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.

A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. See the section "Required Files" below in this chapter.

A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.

The *uuxqt* program comes with a list of commands, */usr/lib/uucp/L.cmds*, that it will execute. A *PATH* shell statement is prepended to the command line as specified in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.

The *L.sys* file should be owned by *uucp* and have mode 0400 to protect the phone numbers and login information for remote sites. (The *uucp*, *uucico*, *uux*, and *uuxqt* should be also owned by *uucp* and have the *setuid* bit set.)

9.9 Installing a Uucp System

The *uucp* system provided with the XENIX Software Development System is already configured for operation on your computer. To install the system, you must edit a few files to provide information about your local site. The following sections provide an overview of the files to be edited and the information required.

During execution of the *uucp* programs, the *uucp* system uses files from the following three directories:

program	(<i>/usr/lib/uucp</i>) This is the directory used for the executable system programs and the system files.
spool	(<i>/usr/spool/uucp</i>) This is the spool directory used during <i>uucp</i> execution.
xqtdir	(<i>/usr/spool/uucp/.XQTDIR</i>) This directory is used during execution of execute files.

The names given in parentheses above are the default values for the directories. The names *lib*, *program*, *xqtdir*, and *spool* will be used in the following text to represent the appropriate directory names.

9.9.1 Modifying the */etc/systemid* File

You must choose a unique site name for each computer to be directly connected to a *uucp* line and add the site name to the */etc/systemid* file of the corresponding computer by using a XENIX text editor. The */etc/systemid* file can actually contain two names: the *uucp* site name, which must appear on the first line of the file, and a Micnet machine name, which must appear on the next line. However, you may decide to have both the *uucp* site name and Micnet machine name to be the same, in which case, only one name is required. For a description of the file, see *systemid(M)* in the XENIX *Reference Manual*.

9.9.2 Creating the Required Files

There are four files which are required for execution, all of which should reside in the *program* directory. To prepare the *uucp* system for execution, you must add your own site specific information to these files by editing the files with a XENIX text editor. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

type line call-unit speed

where *type* must be "ACU" when using an automatic call unit (modem) or "DIR" when using a direct serial line, *line* is the device for the line (e.g. *cul0*), *call-unit* is the automatic call unit associated with *line* (e.g. *cua0*). Hardwired lines have a number "0" in this field, and *speed* is the line speed.

The line

ACU cul0 cua0 300

defines a system which has device "cul0" wired to a call-unit "cua0" for use at 300 baud.

L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. *py*, *mh*, *boston*). The entry format is

abb dial-seq

where *abb* is the abbreviation, and *dial-seq* is the dial sequence to call that location. The line

py 165—

causes the entry *py7777* to be expanded to *165—7777*.

USERFILE

This file contains user accessibility information. It specifies

- The files that can be accessed by a normal user of the local machine
- The files that can be accessed from a remote computer
- The login name used by a particular remote computer
- Whether a remote computer should be called back in order to confirm its identity

Each line in the file has the following format

login,sys [c] pathname [pathname] ...

where *login* is the login name for a user or the remote computer, *sys* is the system name for a remote computer, *c* is the optional call-back required flag, and *pathname* is a pathname prefix that is acceptable for *user*.

It is assumed that the login name used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique system name which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

When the program is obeying a command stored on the local machine, MASTER mode, the pathnames allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

When the program is responding to a command from a remote machine, SLAVE mode, the pathnames allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a null

system name is used.

When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a null system name.

If a line is found that has the appropriate login and remote system names and also contains a “c”, the remote machine is called back before any transactions take place.

The line

```
u,m /usr/xyz
```

allows machine “m” to login with name “u” and request the transfer of files whose names start with “/usr/xyz”.

The line

```
dan, /usr/dan
```

allows the ordinary user “dan” to issue commands for files whose name starts with “/usr/dan”.

The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allow any remote machine to login with name “u”, but if its system name is not “m”, it can only ask to transfer files whose names start with “/usr/spool”.

The lines

```
root, /
, /usr
```

allow any user to transfer files beginning with “/usr” but the user with login “root” can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name The name of the remote system.

time This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800–1730). The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be “Wk” for any week-day, “Any” for any day, or “Never” for special request only. The time should be a range of times (e.g. 0800–1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device This is either “ACU” or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

speed This is the line speed for the call (e.g. 300).

phone The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995–9980). For the hardwired devices, this field contains the same string as used for the device field.

login The login information is given as a series of fields and subfields in the format

expect send [*expect send*] ...

where *expect* is the string expected to be read and *send* is the string to be sent when the expected string is received. The expect field may be made up of subfields of the form

expect [*-send-expect1*] ...

where *send* is sent if the prior *expect* is not successfully read and *expect1* is the next expected string.

There are two special names available to be sent during the login sequence. The string "EOT" sends an EOT character and the string "BREAK" tries to send a BREAK character. (The BREAK character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file is

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm looks at the last part of the string as illustrated in the password field.

9.10 Maintaining the System

This section indicates some events and files which must be maintained for the uucp system. You may do some maintenance with shell command files, initiating the files with *crontab* entries. Others will require manual modification. Some sample shell files are given toward the end of this section.

9.10.1 SEQF – sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

Use of this feature is not recommend.

9.10.2 TM – temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

TM.*pid.ddd*

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

uuclean -pTM

removes all *TM* files older than three days.

9.10.3 LOG – log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

```
uulog
```

performs the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new *LOG* entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

9.10.4 STST – system status files

These files are created in the *spool* directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a talking status when to machines are conversing. The form of the file name is

```
STST.sys
```

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a talking status. In this case, the file must be removed before a conversation is attempted.

9.10.5 LCK – lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

```
LCK..str
```

where *str* is either a device or system name. The files may be left in the *spool* directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

9.10.6 Creating Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a shell file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

```
program /uulog
program /uucico -r1
```

Note that the **-r1** option is required to start the *uucico* program in MASTER mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST*, and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
program /uuclean -pTM -pC. -pD.  
program /uuclean -pST -pLCK -n12
```

can be used. Note the **-n12** option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the **-n** option will use a three day time limit.

9.10.7 Defining Login Entries

One or more logins should be set up for *uucp*. Each of the */etc/passwd* entries should have *program/uucico* as the shell to be executed (where *program* is the directory containing *uucico*). The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the shell argument limits the login to the use of *uucico* only.

9.10.8 Setting File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico*, and *uuxqt* should be owned by the *uucp* login with the setuid bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard shell for the *uucp* logins.

The *L.sys*, *SQFILE*, and *USERFILE* files which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

Chapter 10

The C-Shell

10.1	Introduction	1
10.2	Invoking the C-shell	1
10.3	Using Shell Variables	2
10.4	Using the C-Shell History List	3
10.5	Using Aliases	5
10.6	Redirecting Input and Output	6
10.7	Creating Background and Foreground Jobs	6
10.8	Using Built-In Commands	7
10.9	Creating Command Scripts	8
10.10	Using the argv Variable	8
10.11	Substituting Shell Variables	9
10.12	Using Expressions	10
10.13	Using the C-Shell: A Sample Script	11
10.14	Using Other Control Structures	13
10.15	Supplying Input to Commands	13
10.16	Catching Interrupts	14
10.17	Using Other Features	14
10.18	Starting a Loop at a Terminal	14
10.19	Using Braces with Arguments	15
10.20	Substituting Commands	16
10.21	Special Characters	16

10.1 Introduction

The C-shell program, *cs**h*, is a command language interpreter for XENIX system users. The C-shell, like the standard XENIX shell *sh*, is an interface between you and the XENIX commands and programs. It translates command lines typed at a terminal into corresponding system actions, gives you access to information, such as your login name, home directory, and mailbox, and lets you construct shell procedures for automating system tasks.

This chapter explains how to use the C-shell. It also explains the syntax and function of C-shell commands and features, and shows how to use these features to create shell procedures. The C-shell is fully described in *cs**h*(CP) in the XENIX *Reference Manual*.

10.2 Invoking the C-shell

You can invoke the C-shell from another shell by using the **cs***h* command. To invoke the C-shell, type:

```
cs
```

at the standard shell's command line. You can also direct the system to invoke the C-shell for you when you log in. If you have given the C-shell as your login shell in your */etc/passwd* file entry, the system automatically starts the shell when you log in.

After the system starts the C-shell, the shell searches your home directory for the command files *.cshrc* and *.login*. If the shell finds the files, it executes the commands contained in them, then displays the C-shell prompt.

The *.cshrc* file typically contains the commands you wish to execute each time you start a C-shell, and the *.login* file contains the commands you wish to execute after logging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C-shell; there is no corresponding XENIX program for this command. **Set** sets the C-shell variable "ignoreeof" which shields the C-shell from logging out if CNTRL-D is hit. Instead of CNTRL-D, the **logout** command is used to log out of the system. By setting the "mail" variable, the C-shell is notified that it is to watch for incoming mail and notify you if new mail arrives.

Next the C-shell variable "time" is set to 15 causing the C-shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the C-shell will remember the last 10 commands typed in its history list, (described later).

Finally, the XENIX *mail* program is invoked.

When the C-shell finishes processing the *.login* file, it begins reading commands from the terminal, prompting for each with:

```
%
```

When you log out (by giving the **logout** command) the C-shell prints

```
logout
```

and executes commands from the file *.logout* if it exists in your home directory. After that, the C-shell terminates and XENIX logs you off the system.

10.3 Using Shell Variables

The C-shell maintains a set of variables. For example, in the above discussion, the variables "history" and "time" had the values 10 and 15. Each C-shell variable has as its value an array of zero or more strings. C-shell variables may be assigned values by the **set** command, which has several forms, the most useful of which is:

```
set name=value
```

C-shell variables may be used to store values that are to be used later in commands through a substitution mechanism. The C-shell variables most commonly referenced are, however, those that the C-shell itself refers to. By changing the values of these variables you can directly affect the behavior of the C-shell.

One of the most important variables is "path". This variable contains a list of directory names. When you type a command name at your terminal, the C-shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you typed. The **set** command with no arguments displays the values of all variables currently defined in the C-shell. The following example shows a typical default values:

```
argv      ()
home      /usr/bill
path      (. /bin /usr/bin)
prompt    %
shell     /bin/csh
status    0
```

This output indicates that the variable "path" begins with the current directory indicated by dot (.), then */bin*, and */usr/bin*. Your own local commands may be in the current directory. Normal XENIX commands reside in */bin* and */usr/bin*.

Sometimes a number of locally developed programs reside in the directory */usr/local*. If you want all C-shells that you invoke to have access to these new programs, place the command

```
set path=(. /bin /usr/bin /usr/local)
```

in the *.cshrc* file in your home directory. Try doing this, then re-executing you *.login* with the command **source.login**. Type

```
set
```

to see that the value assigned to "path" has changed.

You should be aware that when you log in the C-shell examines each directory that you insert into your path and determines which commands are contained there, except for the current directory which the C-shell treats specially. This means that if commands are added to a directory in your search path after you have started the C-shell, they will not necessarily be found. If you wish to use a command which has been added after you have logged in, you should give the command

```
rehash
```

to the C-shell. **Rehash** causes the shell to recompute its internal table of command locations, so that it will find the newly added command. Since the C-shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built in variables are "home" which shows your home directory, and "ignoreeof" which can be set in your *.login* file to tell the C-shell not to exit when it receives an end-of-file from a terminal. The variable "ignoreeof" is one of several variables whose value the C-shell does not care about; the C-shell is only concerned with whether these variables are set or unset. Thus, to set "ignoreeof" you simply type

```
set ignoreeof
```

and to unset it type

```
unset ignoreeof
```

Some other useful built-in C-shell variables are “noclobber” and “mail”. The syntax

```
> filename
```

which redirects the standard output of a command just as in the regular shell, overwrites and destroys the previous contents of the named file. In this way, you may accidentally overwrite a file which is valuable. If you prefer that the C-shell not overwrite files in this way you can

```
set noclobber
```

in your *.login* file. typing

```
date > now
```

causes an error message if the file *now* already exists. You can type

```
date >! now
```

if you really want to overwrite the contents of *now*. The “>!” is a special syntax indicating that overwriting or “clobbering” the file is ok. (The space between the exclamation point (!) and the word “now” is critical here, as “!now” would be an invocation of the history mechanism, described below, and have a totally different effect.)

10.4 Using the C-Shell History List

The C-shell can maintain a history list into which it places the text of previous commands. It is possible to use a notation that reuses commands, or words from commands, in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the C-shell. Boldface indicates user input:


```

% cat bug.c
main()

{
    printf("hello");
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/" & /p
    printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
    printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill      3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill      3932 Dec 19 09:42 bug
% bug
hello
% pr bug.c | lpt
lpt: Command not found.
% ^lpt^lpr
pr bug.c | lpr
%
```

In this example, we have a very simple C program that has a bug or two in the file *bug.c*, which we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as “!\$”, meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign (\$) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end-of-line. The C-shell echoed the command, as it would have been typed without use of the history mechanism, and then executed the command. The compilation yielded error diagnostics, so we now edit the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as “!c”, which repeats the last command that started with the letter “c”. If there were other commands beginning with the letter “c” executed recently, we could have said “!cc” or even “!cc:p” which prints the last command starting with “cc” without executing it, so

that you can check to see whether you really want to execute a given command.

After this recompilation, we ran the resulting *a.out* file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra “-o bug” telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. In general, the history mechanisms may be used anywhere in the formation of new commands, and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then we ran an “ls -l” command with the same argument list, denoting the argument list:

!*
 *

Finally, we ran the program *bug* to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file *bug.c*. In order to print the listing at a lineprinter we piped the output to **lpr**, but misspelled it as “lpt”. To correct this we used a C-shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with

!!
 !!

and sent its output to the lineprinter.

There are other mechanisms available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in *cs*h(CP) the XENIX *Reference Manual*.

10.5 Using Aliases

The C-shell has an alias mechanism that can be used to make transformations on commands immediately after they are input. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained by using C-shell command files, but these take place in another instance of the C-shell and cannot directly affect the current C-shell’s environment or involve commands such as **cd** which must be done in the current C-shell.

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program *mail*. If you place the C-shell command

alias mail newmail

in your *.cshrc* file, the C-shell will transform an input line of the form

mail bill

into a call on *newmail*. Suppose you wish the command **ls** to always show sizes of files, that is, to always use the **-s** option. In this case, you can use the **alias** command to do

alias ls ls -s

or even

alias dir ls -s

creating a new command named **dir**. If we then type

dir ~bill

the C-shell translates this to


```
ls -s /usr/bill
```

Note that the tilde (~) is a special C-shell symbol that represents the user's home directory.

Thus the **alias** command can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

specifies an **ls** command after each **cd** command. We enclosed the entire alias definition in single quotation marks (') to prevent more substitutions from occurring and to prevent the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The "\!" here substitutes the entire argument list to the prealiasing **cd** command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly the following example defines a command that looks up its first argument in the password file.

```
alias whois 'grep \!^ /etc/passwd'
```

The C-shell currently reads the *.cshrc* file each time it starts up. If you place a large number of aliases there, C-shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15 is reasonable). Too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

10.6 Redirecting Input and Output

In addition to the standard output, commands also have a diagnostic output that is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally useful to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can type

```
command > & file
```

The "> &" here tells the C-shell to route both the diagnostic output and the standard output into *file*. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the lineprinter. The form

```
command > &! file
```

is used when "noclobber" is set and *file* already exists.

Finally, use the form

```
command >> file
```

to append output to the end of an existing file. If "noclobber" is set, then an error results if *file* does not exist, otherwise the C-shell creates *file*. The form

```
command >>! file
```

lets you append to a file even if it does not exist and "noclobber" is set.

10.7 Creating Background and Foreground Jobs

When one or more commands are typed together as a pipeline or as a sequence of commands

separated by semicolons, a single job is created by the C-shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the C-shell creates a job. Each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is typed at the end of the commands, then the job is started as a background job. This means that the C-shell does not wait for the job to finish, but instead, immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C-shell. Thus

```
du > usage &
```

runs the *du* program, which reports on the disk usage of your working directory, puts the output into the file *usage* and returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program continues executing in the background until it finishes, even though you can type and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard such as the INTERRUPT or QUIT signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job you want killed. Process numbers can be found with the **ps** command.

10.8 Using Built-In Commands

This section explains how to use some of the built-in C-shell commands.

The **alias** command described above is used to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. It may also be given one argument, such as to show the current alias for a given string of characters. For example

```
alias ls
```

prints the current alias for the string "ls".

The **history** command displays the contents of the history list. The numbers given with the history events can be used to reference previous events that are difficult to reference contextually. There is also a C-shell variable named "prompt". By placing an exclamation point (!) in its value the C-shell will substitute the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example, you could type:

```
set prompt='\! % '
```

Note that the exclamation mark (!) had to be escaped even within backslashes.

The **logout** command is used to terminate a login C-shell that has "ignoreeof" set.

The **rehash** command causes the C-shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C-shell's search path and want the C-shell to find it, since otherwise the hashing algorithm may tell the C-shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command is used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could type:

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

sets the value of the environment variable "TERM" to "adm3a". The program *env* exists to print

out the environment. For example, its output might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

The **source** command is used to force the current C-shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file that you wish to take effect before the next time you login.

The **time** command is used to cause a command to be timed no matter how much CPU time it takes. Thus

```
time cp /etc/rc /usr/bill/rc
```

displays:

```
0.0u 0.1s 0:01 8%
```

Similarly

```
time wc /etc/rc /usr/bill/rc
```

displays:

```
52 178 1347 /etc/rc
52 178 1347 /usr/bill/rc
104 356 2694 total
0.1u 0.1s 0:00 13%
```

This indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01). The word count command **wc** used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the **wc** command used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands are used to remove aliases and variable definitions from the C-shell.

10.9 Creating Command Scripts

It is possible to place commands in files and to cause C-shells to be invoked to read and execute commands from these files, which are called C-shell scripts. This section describes the C-shell features that are useful when creating C-shell scripts.

10.10 Using the argv Variable

A **csh** command script may be interpreted by saying

```
csh script argument ...
```

where *script* is the name of the file containing a group of C-shell commands and *argument* is a sequence of command arguments. The C-shell places these arguments in the variable "argv" and then begins to read commands from *script*. These parameters are then available through the same mechanisms that are used to reference any other C-shell variables.

If you make the file *script* executable by doing

```
chmod 755 script
```

or

```
chmod +x script
```

and then place a C-shell comment at the beginning of the C-shell script (i.e., begin the file with a number sign (#)) then */bin/csh* will automatically be invoked to execute *script* when you type

```
script
```

If the file does not begin with a number sign (#) then the standard shell */bin/sh* will be used to execute it.

10.11 Substituting Shell Variables

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign (\$), this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable “argv” to be echoed to the output of the C-shell script. It is an error for “argv” to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to 1 if *name* is set or to 0 if *name* is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable “name”. To illustrate, examine the following terminal session (input is in boldface):

```
% set argv=(a b c)
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable that has several values. Thus

```
$argv[1]
```

gives the first component of “argv” or in the example above “a”. Similarly

```
$argv[$#argv]
```

would give “c”, and

```
$argv[1-2]
```

would give:

a b

Other notations useful in C-shell scripts are

$\$n$

where n is an integer. This is shorthand for

$\$argv[n]$

the n 'th parameter and

$\$*$

which is a shorthand for

$\$argv$

The form

$\$\$$

expands to the process number of the current C-shell. Since this process number is unique in the system, it is often used in the generation of unique temporary filenames. The form

$\$<$

is quite special and is replaced by the next line of input read from the C-shell's standard input (not the script it is reading). This is useful for writing C-shell scripts that are interactive, reading commands from the terminal, or even writing a C-shell script that acts as a filter, reading lines from its input file. Thus, the sequence

```
echo -n 'yes or no?'  
set a=($<)
```

writes out the prompt

yes or no?

without a newline and then reads the answer into the variable "a". In this case "\$#a" is 0 if either a blank line or CNTRL-D is typed.

One minor difference between " $\$n$ " and " $\$argv[n]$ " should be noted here. The form " $\$argv[n]$ " will yield an error if n is not in the range 1- $\$#argv$ while " $\$n$ " will never yield an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

Another important point is that it is never an error to give a subrange of the form " $n-$ "; if there are less than " n " components of the given variable then no words are substituted. A range of the form " $m-n$ " likewise returns an empty vector without giving an error when " m " exceeds the number of elements of the given variable, provided the subscript " n " is in range.

10.12 Using Expressions

To construct useful C-shell scripts, the C-shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C-shell with the same precedence that they have in C. In particular, the operations " $==$ " and " $!=$ " compare strings and the operators " $\&\&$ " and " $\|\|$ " implement the logical AND and OR operations. The special operators " $=~$ " and " $!~$ " are similar to " $==$ " and " $!=$ " except that the string on the right side can have pattern matching characters (like $*$, $?$ or $[$ and $]$). These operators test whether the string on the left matches the pattern on the right.

The C-shell also allows file enquiries of the form

$-? filename$

where question mark (?) is replaced by a number of single characters. For example, the

expression primitive

`-e filename`

tells whether *filename* exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form

`{ command }`

which returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with exit status nonzero. If more detailed information about the execution status of a command is required, it can be executed and the “status” variable examined in the next command. Since “\$status” is set by every command, its value is always changing.

For the full list of expression components, see `cs(1P)` in the XENIX *Reference Manual*.

10.13 Using the C-Shell: A Sample Script

A sample C-shell script follows that uses the expression mechanism of the C-shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

    if ($i != *.c) continue # not a .c file so do nothing

    if (! -r ~/backup/$i:t) then
        echo $i:t not in backup... not cp'ed
        continue
    endif

    cmp -s $i ~/backup/$i:t # to set $status

    if ($status != 0) then
        echo new backup of $i
        cp $i ~/backup/$i:t
    endif
end
```

This script uses the **foreach** command. The command executes the other commands between the **foreach** and the matching **end**. for each of the values given between parentheses with the named variable “i” which is set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (*i* in this case) has the value at the last iteration.

The “noglob” variable is set to prevent filename expansion of the members of “argv”. This is a good idea, in general, if the arguments to a C-shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a “\$” variable expansion, but this is harder and less reliable.

The other control construct is a statement of the form


```

if ( expression ) then
    command
    ...
endif

```

The placement of the keywords in this statement is not flexible due to the current implementation of the C-shell. The following two formats are not acceptable to the C-shell:

```

if (expression) # Won't work!
then
    command
    ...
endif

```

and

```

if (expression) then command endif # Won't work

```

The C-shell does have another form of the if statement:

```

if ( expression ) command

```

which can be written

```

if ( expression ) \
    command

```

Here we have escaped the newline for the sake of appearance. The command must not involve “|”, “&” or “;” and must not be another control command. The second form requires the final backslash (\) to immediately precede the end-of-line.

The more general **if** statements above also admit a sequence of **else—if** pairs followed by a single **else** and an **endif**, for example:

```

if ( expression ) then
    commands
else if (expression ) then
    commands
...

else
    commands
endif

```

Another important mechanism used in C-shell scripts is the colon (:) modifier. We can use the modifier **:r** here to extract the root of a filename or **:e** to extract the extension. Thus if the variable “i” has the value */mnt/foo.bar* then

```

echo $i $i:r $i:e

```

produces

```

/mnt/foo.bar /mnt/foo bar

```

This example shows how the **:r** modifier strips off the trailing “.bar” and the **:e** modifier leaves only the “bar”. Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **cs**h(CP) entry in the *XENIX Reference Manual*. It is also possible to use the command substitution mechanism to perform modifications on strings to then reenter the C-shell environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C-shell limits the number of colon modifiers on a “\$” substitution to 1. Thus

```
% echo $i $i:h:t
```

produces

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, we note that the number sign character (#) lexically introduces a C-shell comment in C-shell scripts (but not from the terminal). All subsequent characters on the input line after a number sign are discarded by the C-shell. This character can be quoted using `"#"` or `"#"` argument word.

10.14 Using Other Control Structures

The C-shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
    commands
end
```

and

```
switch ( word )

case str1:
    commands
    breaksw

...

case strn:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

For details see the manual section for **csh**(CP). C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in C-shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C-shell allows a **goto** statement, with labels looking like they do in C:

```
loop:
    commands
    goto loop
```

10.15 Supplying Input to Commands

Commands run from C-shell scripts receive by default the standard input of the C-shell which is running the script. It allows C-shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

Thus we need a metanotation for supplying inline data to commands in C-shell scripts. For example, consider this script which runs the editor to delete leading blanks from the lines in each

argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation

```
<< 'EOF'
```

means that the standard input for the **ed** command is to come from the text in the C-shell script file up to the next line consisting of exactly EOF. The fact that the EOF is enclosed in single quotation marks ('), i.e., it is quoted, causes the C-shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the C-shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form "1,\$" in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign (\$) with a backslash (\), i.e.:

```
1,\$s/^[ ]*//
```

Quoting the EOF terminator is a more reliable way of achieving the same thing.

10.16 Catching Interrupts

If our C-shell script creates temporary files, we may wish to catch interruptions of the C-shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the C-shell will do a "goto label" and we can remove the temporary files, then do an **exit** command (which is built in to the C-shell) to exit from the C-shell script. If we wish to exit with nonzero status we can write

```
exit (1)
```

to exit with status 1.

10.17 Using Other Features

There are other features of the C-shell useful to writers of C-shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can be used to help trace the actions of the C-shell. The **-n** option causes the C-shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that the C-shell will not execute C-shell scripts that do not begin with the number sign character (#), that is C-shell scripts that do not begin with a comment.

There is also another quotation mechanism using the double quotation mark ("), which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as the single quote (') does.

10.18 Starting a Loop at a Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a

number of similar commands. For instance, if there were three shells in use on a particular system, */bin/sh*, */bin/nsh*, and */bin/csh*, you could count the number of persons using each shell by using the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v sh$ /etc/passwd
```

Since these commands are very similar we can use **foreach** to simplify them:

```
$ foreach i ('csh$' 'nsh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C-shell prompts for input with “?” when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Also useful with loops are variables that contain lists of filenames or other words. For example, examine the following terminal session:

```
% set a=('ls')
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

The **set** command here gave the variable “a” a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within back quotation marks (‘’) is converted by the C-shell to a list of words. You can also place the quoted string within double quotation marks (”) to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier :x exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

10.19 Using Braces with Arguments

Another form of filename expansion involves the characters, “{” and “}”. These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e., nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/ {hdrs,retrofit,csh}
```

to make subdirectories *hdrs*, *retrofit* and *csh* in your home directory. This mechanism is most useful when the common prefix is longer than in this example:


```
chown root /usr/demo/{file1,file2,...}
```

10.20 Substituting Commands

A command enclosed in accent symbols (```) is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable “pwd” or to do

```
vi `grep -l TRACE *.c`
```

to run the editor `vi` supplying as arguments those files whose names end in `.c` which have the string “TRACE” in them. Command expansion also occurs in input redirected with “<<” and within quotation marks (`"`). Refer to `cs`h(CP) in the *XENIX Reference Manual* for more information.

10.21 Special Characters

The following table lists the special characters of `cs`h and the XENIX system. A number of these characters also have special meaning in expressions. See the `cs`h manual section for a complete list.

Syntactic metacharacters

- `;` Separates commands to be executed sequentially
- `|` Separates commands in a pipeline
- `()` Brackets expressions and variable values
- `&` Follows commands to be executed without waiting for completion

Filename metacharacters

- `/` Separates components of a file's pathname
- `.` Separates root parts of a filename from extensions
- `?` Expansion character matching any single character
- `*` Expansion character matching any sequence of characters
- `[]` Expansion sequence matching any single character from a set of characters
- `~` Used at the beginning of a filename to indicate home directories
- `{ }` Used to specify groups of arguments with common parts

Quotation metacharacters

- `\` Prevents meta-meaning of following single character
- `'` Prevents meta-meaning of a group of characters
- ``` Like `'`, but allows variable and command expansion

Input/output metacharacters

< Indicates redirected input

> Indicates redirected output

Expansion/Substitution Metacharacters

\$ Indicates variable substitution

! Indicates history substitution

: Precedes substitution modifiers

^ Used in special forms of history substitution

` Indicates command substitution

Other Metacharacters

Begins scratch filenames; indicates C-shell comments

- Prefixes option (flag) arguments to commands

Chapter 11

Using The Visual Shell

11.1	What is the Visual Shell?	1
11.2	Getting Started with the Visual Shell	1
11.2.1	Entering the Visual Shell	1
11.2.2	Getting Help	2
11.2.3	Leaving the Visual Shell	2
11.3	The Visual Shell Screen	2
11.3.1	Status Line	2
11.3.2	Message Line	2
11.3.3	Main Menu	2
11.3.4	Command Option Menu	2
11.3.5	Program Output	3
11.3.6	View Window	3
11.4	Visual Shell Reference	4
11.4.1	Visual Shell Default Menu	4
11.4.2	Options	6
11.4.3	Print	7
11.4.4	Quit	7
11.4.5	Run	7
11.4.6	View	8
11.4.7	Window	8
11.4.8	Pipes	8
11.4.9	Count	8
11.4.10	Get	8
11.4.11	Head	9
11.4.12	More	9
11.4.13	Run	9
11.4.14	Sort	9
11.4.15	Tail	9

11.1 What is the Visual Shell?

The Visual Shell **vsh** is a menu-driven XENIX shell. This chapter describes the use and behavior of the **vsh**. This chapter assumes that the reader is familiar with some general XENIX concepts, specifically the structure of XENIX filesystems and the nature of a XENIX 'command'. No familiarity with any other shell, however, is assumed. If you are a first-time user of the Visual Shell, please completely read the narrative sections of this chapter.

A 'shell' is a program which passes a command to an operating system, and displays the result of running the command. The XENIX shells can also create 'pipelines' for passing the output of one command to another command or 'redirect' the output into a file.

The other XENIX shells available are **sh** and **cs**. These shells are called 'command-line oriented' shells. This means that the user enters commands one line at a time. The **sh** and **cs** shells are full computer languages which require study and some programming knowledge to use effectively. These command-line shells are powerful and efficient.

The **vsh** is a 'menu-oriented' shell. In a menu-oriented shell, the user is given the available commands, or some of the available commands. The user can run the command, by selecting from the menu.

The Visual Shell is a good shell for users who may not want to master a programming language right away just to use XENIX or a specific XENIX application. All Visual Shell users should additionally become familiar with some command-line shell usage.

Users familiar with command-line shells are in for a pleasant surprise if they try the Visual Shell. Experienced users will appreciate the efficiency and versatility of the Visual Shell. The distinction is very much akin to the difference between a line-oriented text editor and a full-screen editor.

A menu shell can be used effectively with very little study. On the other hand, a menu shell can also restrict the user from using the operating system in creative, possibly more efficient ways. The Microsoft Visual Shell strikes a balance in this regard. The Visual Shell is designed to do all of the things that the command-line shells can do.

11.2 Getting Started with the Visual Shell

This section describes how to enter, obtain help about, and leave the visual shell. This section also describes what you will see on the screen while running the visual shell and how the menus work.

Note the following convention for specifying keystrokes. CTRL refers to the CTRL shift key. CTRL-C means pressing the CTRL and 'c' keys at the same time. Note the irrelevance of case in entering Menu Selection characters. For instance, press either 'Q' or 'q' to run the "Quit" command from the main menu.

11.2.1 Entering the Visual Shell

Log in to XENIX. If you are not sure how to log in, consult the Operations Guide or have someone knowledgeable about XENIX help you. When you have a shell prompt (typically '\$' or '%'), the operating system is waiting for a command. Enter the command:

vsh

and press RETURN.

11.2.2 Getting Help

If at anytime you are not sure what to do, either run the "Help" Menu Selection. Refer to the reference section of this chapter for information about the Help command.

11.2.3 Leaving the Visual Shell

To exit the Visual Shell select the Quit command from the main menu. The simplest way to do this is to simply press 'q' or 'Q'. In response to the prompt "Type Y to confirm", enter 'y' or 'Y'. If you don't want to exit the Visual Shell yet (perhaps you pressed 'q' by mistake), enter any other character but 'y' or 'Y'. If you have invoked the visual shell from another shell, as described above, you will need to log out from XENIX by entering CTRL-D or 'logout' and pressing RETURN. If the Visual Shell is your default shell, you will automatically be logged out.

11.3 The Visual Shell Screen

11.3.1 Status Line

The bottom line on the screen is called the 'status line'. The status line displays the name of the current working directory, notifies you if you have mail, and gives the date, time and the name of the operating system.

11.3.2 Message Line

The line above the 'status line' is called the 'message line'. The message line displays special output from XENIX commands, such as error reports.

11.3.3 Main Menu

The next section of the screen above the message line is the 'main menu'. The main menu displays a selection of useful XENIX commands.

The currently selected menu command is highlighted on the screen. To select any command, press the SPACE BAR. The next highlighted command is selected. The BACKSPACE key will move to the previous command. Move through the menu until you have found the command you want. To run the currently selected command, press RETURN.

You may also enter the first letter of a command to select that command. If you enter the first letter of the command, you do not need to press RETURN.

If you enter a letter which does not correspond to a menu selection, the message

Not a valid option

will be displayed. Try another option.

11.3.4 Command Option Menu

When you have selected a command, the main menu will be replaced with a command option menu. The command option menu gives the options available with the specific command. You must fill in the options with appropriate responses.

If you wish to return to the main menu without running the command, press CTRL-C, (cancel). If you want to run the command with the selected options press RETURN.

The following keystrokes allow editing of option responses.

CTRL-I, or 'tab'	Move to next field in options menu.
CTRL-Y or DEL	Delete character under cursor.
CTRL-L	Move cursor to character to right of current position in current option field.
CTRL-K	Move cursor to character to left of current position in current option field.
CTRL-P	Move cursor to word in current field to right of the current word.
CTRL-O	Move cursor to word in current field to left of the current word.

11.3.5 Program Output

While running a command, commands given and output (unless redirected) will be displayed above the menu and below the view window. The output *scrolls up*: moves from bottom to top. Lines scrolling off the top of the output window disappear.

Visual Shell command lines are listed with each argument preceded by the number in the argument list enclosed in parentheses. The command is named in the output window by the menu command. Hence, if you run the command `/bin/ls` with the argument `-R`, the output window will display the command line as follows:

Run (1) /bin/ls (2) -R

To change the command line format to reflect the actual XENIX command line generated by the Visual Shell, use the Options Output menu command.

11.3.6 View Window

A menu of currently accessible files and directories can be displayed at the top of the screen in alphabetical order, left to right, top to bottom. Note that this display is the same as that obtained using the view command. This will be referred to as the 'view window' in this chapter. If the directory list is larger than the current window size, you may scroll through using the key commands given below. To reset the window size, use the 'Window' main menu command.

The currently selected item is highlighted in the view window. Use the arrow keys and other key commands given at the end of this section to move the highlight around the window.

If a directory is being listed, subdirectories are shown enclosed in square brackets. To view a subdirectory, press '=' while the directory is highlighted. To return to the previous directory after viewing a subdirectory, press '-'. The parent directory of the current directory is shown as '[.]'. The current directory is shown as '[.]'. Executable files are preceded by an asterisk. The last modification date of the currently selected item is given at the right margin of the last line of the window. The name of the item in view in the current window is given in the upper right-hand corner of the window.

The view window may also display contents of files. Highlight a file, and press '='. You may scroll through the file using the key commands given below. While viewing a file, the highlighted area covers one line.

If you press '=' while an executable file is highlighted, that file will be run.

If the Visual Shell requires a file or directory name, the currently selected View Window item can be automatically entered in the relevant option field by pressing any directional movement key

following selection of the command. This method saves keystrokes and reduces the chance of making typing mistakes. On the other hand, if you wish to explicitly enter a file or directory in an option field, type in the name after selecting the command.

Use these keystrokes to select files from the view window:

WINDOW MOTION KEYS

CTRL-Q	Move to start (first item alphabetically) of view window.
CTRL-Z	Move to end (last item alphabetically) of view window.
CTRL-R CTRL-E	Scroll view window up.
CTRL-R CTRL-X	Scroll view window down.
=====	View indicated item, either file or directory.
	If no view window is present, the current working directory is displayed.
	Return window display to parent directory of currently listed directory.
	If viewing a file, exit from viewing that file.
	Last view window is returned to.

DIRECTIONAL MOVEMENT KEYS

ARROW UP or CTRL-E:	Move highlight up in view window.
ARROW DOWN or CTRL-X:	Move highlight down in view window.
ARROW LEFT or CTRL-S:	Move highlight left in view window.
ARROW RIGHT or CTRL-D:	Move highlight right in view window.

Movement beyond the left or right margin will proceed to the next item on the previous or next line unless at the edge of the view window. Movement beyond the top or bottom edge of the current window will scroll the view window up or down if there are more items in that direction in the view window.

Note that there are two ways to move the highlight around. Either use the keypad arrow keys or the cluster of four keys on the far left of the keyboard 'e', 'x', 's', and 'd' shifted with CTRL.

While viewing a file, the directional movement keys for up and left move the highlight up, and the keys for down and right move the highlighted line down.

11.4 Visual Shell Reference

11.4.1 Visual Shell Default Menu

This section describes the default Visual Shell menu commands and options. The menu options are displayed at the bottom of the screen above the status line.

To invoke a command, move the highlight forwards through the main menu using the space bar or the tab key, or backwards using the backspace key. Or simply press the first letter of the command.

Most commands require entering options. Move the cursor to the field using the SPACE BAR, TAB key or BACKSPACE key, and type your response. To edit the options, refer to the key commands listed above in the section in this chapter labelled "Command Option Menu". To select an item from a View Window listing for insertion in a field, refer to the section in this chapter labelled "View Window".

Note that some options have 'switches' with predefined (default) selections. The currently selected switch setting is highlighted. The default is the parenthesized setting. For instance, in the switch:

Recursive: (yes) no

the default is recursive. To change a switch, select the field and press the SPACE BAR or BACKSPACE.

Copy

The Copy command can copy files and directories. To copy a file, select "File" from the options, to copy a directory, select "Directory". A sub-menu will appear. Enter the file or directory you wish copied in the *from:* field. Enter the file or directory you wish copied to the *to:* field. Note that if the item in the *to:* field already exists, it will be overwritten, so be careful.

The Copy Directory sub-menu has a switch "recursive". If this switch is set to yes, all sub-directories and their contents below the specified directory will be copied.

Delete

The Delete command can remove files and directories. In the *DELETE*name: field enter the name of the file or directory you want to remove. Note that once the file or directory is deleted, the contents are gone forever unless you have another copy, so be careful.

Edit

The Edit command invokes the full-screen editor *vi*. The current directory will be displayed in the output window. Enter in the option field *EDIT filename:* the name of the file you wish to edit using *vi*.

To learn *vi*, consult the document "vi: a Screen Editor" in the *XENIX User's Guide*, and the *vi*(C) manual page in the *XENIX Reference*. A *vi* reference card is also available.

Do not use a "-" in file names since it is a reserved character of *vsh*.

Help

The Help command can give on-line help regarding many aspects of Visual Shell use. The view window will display the help file. Use the menu to select the topic you need help with. For instance, move the highlight to 'Keyboard' using the SPACE BAR and press RETURN to view the help file starting at the 'Keyboard' section. The 'Next' and 'Previous' fields in the menu will scroll through the the help file from the present location one screen at a time. Your work will remain undisturbed. To return from Help, press CTRL-C or select the 'Resume' menu option.

Mail

The Mail command enters the XENIX mail system. There are two options: "Send" and "Read". For more information about mail, refer to the section of the *XENIX Users Guide* titled "Mail" or

refer to the **mail(C)** manual page.

Name

The Name command renames an existing file or directory. There are two fields, *From:* and *To:*. Enter the name of the file or directory you want to rename in *From:* and the new name in *To:*

11.4.2 Options

The Options Main Menu Selection provides four sub-menus. These sub-menus run commands which typically are used infrequently or which have irrevocable results.

Directory Option

The Directory command has two sub-menus, Make and Usage.

Make Directory Option

This command creates a new directory named what you enter in the *name:* field.

Usage Directory Option

Counts the number of disk blocks in the directories specified in the *name:* field. The format is the same as the XENIX command **ls -s**. Refer to the manual page **ls(C)**.

FileSystem Option

FileSystem has the five sub-menus: Create, FilesCheck, SpaceFree, Mount and Unmount.

Create FileSystem Option

Create FileSystem makes a XENIX filesystem. The Create command performs radical system maintenance and may have irrevocable effects. Care is advised when using Create FileSystem.

The functionality is the same as **mkfs(C)**. Consult the **mkfs(C)** manual page before running Create FileSystem. Create FileSystem will prompt you for device, block size, gap number and block number. Refer to the XENIX *Operations Guide* chapter on "Using File Systems". The section "Creating a File System" also explains this command.

FilesCheck FileSystem Option

FilesCheck checks the consistency of a XENIX filesystem and attempts repair if damage is

detected. The FilesCheck command performs radical system maintenance and may have irrevocable effects. Care is advised when using FilesCheck.

The functionality is the same as **fsck(C)**. Consult the **fsck(C)** manual page before running FilesCheck. FilesCheck will prompt you for the device to check.

Output Option

The Output Option command has one switch, *commands like: VShell XENIX*". The default is VShell. IF VShell is set, the **vsh** form of commands given appear in the upward scrolling output window. If XENIX is specified, the XENIX command line which **vsh** generated will be shown instead.

Permissions Option

The Permissions Option command allows changing the access permissions on files and directories. The functionality is the same as the **chmod(C)** command. Consult the **chmod** manual page if you do not understand the concept of XENIX permissions.

In the *name:* field enter the name of the file or directory you wish to alter the permissions on. You may only alter the permissions on files and directories you own. There are four switches, *who:*, *read:*, *write:*, and *execute:*.

The *who:* switch has four settings, *All*, *Me*, *Group* and *Others*. *All* is the default. *All* refers to yourself, those with the same group id as yourself and others. *Me* refers to yourself. *Group* refers to all others with your group id. *Others* refers to those outside your group.

The read, write and execute switches have two settings, yes and no. The default is yes for *Me*, and no for *Group* and *Others*. This grants the given type of permission to those specified in the *who:* switch. No takes away the given type of permission from those specified in the *who:* switch.

11.4.3 Print

The Print command puts a file or files in the queue for your lineprinter. In the *filename:* option field, enter the file or files you want to print.

11.4.4 Quit

The Quit command exits the Visual Shell. The only option is *Enter Y to confirm:*. Enter 'Y' or 'y' if you really want to quit. Any other key cancels the quit.

11.4.5 Run

The Run command executes a program or shell script. The *name:* option takes the name of an executable file. In the *parameters:* option field enter flags to pass to the executable file. The *output:* option can specify a file to redirect output to or another program to send the output to. Enter a vertical bar '↑' in the output field to use the pipe menu.

It is also possible to run an executable file by highlighting the name of the file in the View Window and pressing '='.

11.4.6 View

The View command allows you to inspect without altering the contents of files and directories. View is also available at any time for an item highlighted in the View Window by pressing '='. See the section above labelled 'View Window' for the details of using View.

To alter the height and characteristics of the View Window, use the 'Window' menu option. See the section below labelled "Window".

If you have invoked View from the menu, enter the name of the file or directory you wish to view in the *VIEW name:* field, or select from a directory view window.

To return from any View action to the previously displayed View Window, press the minus key '-'.

If you View a non-executable binary file, non-ascii characters are displayed as the character '@'.

11.4.7 Window

The Window command alters the height and redraw characteristics of the Visual Shell View Window.

The

WINDOW redraw: Yes (No)

switch turns on or off redraw of the view window after running a command.

The *heightinlines:* field changes the number of lines displayed in the view window. The minimum window height is 1 lines. The default window height is 5 lines. The maximum window height is 15 lines.

11.4.8 Pipes

XENIX allows output from one program to be passed to another program or to be put in a file. This is called 'piping' or 'pipelining'. If the output is placed in a file it is said to be 'redirected'. Piping is supported in the Visual Shell through the pipe menu.

The Pipe menu is invoked by entering a vertical bar '|' character in any option field named *output:*. For instance, the Run main menu and the Pipe menu itself have an *output:* field. The available Pipe menu commands are Count, Get, Head, More, Run, Sort and Tail. Each Pipe menu sub-command also has an *output:* field, which allows construction of pipelines of arbitrary length.

11.4.9 Count

Count counts words, lines and characters in the input pipe. The default is all of the above. There is a switch for each type of item to count. The Count Pipe Menu option corresponds to the XENIX command **wc**. Consult the manual page **wc(C)** for the functionality.

11.4.10 Get

Get looks for patterns in the input pipe. The pattern may be verbatim, or you may specify a "regular expression" to look for. Regular expressions may contain 'wildcard' characters which represent sets of strings. Consult the manual page **grep(C)** for the available wildcard characters.

The first Get switch is *Unmatched (Yes) No*. If you specify Yes (the default), all lines containing the given pattern will be output. If Unmatched is set to off, all lines not containing the given

pattern will be output.

The second Get switch is *ignore case*: which suppresses the case while looking for the regular expression. The default is off.

The third Get switch is *line numbers*:, which reports the line in the input stream which the regular expression was matched on. The default is on.

11.4.11 Head

Head prints a specified number of lines of the input stream starting from the first line. The *lines*: field may be set to specify the number of lines at the head of the input stream to print. The default is 5 lines.

The Head Pipe Menu option corresponds to the XENIX command head. Consult the manual page **head(C)** for the functionality.

11.4.12 More

More allows viewing an input stream one screen at a time. The More Pipe Menu option invokes the XENIX command more. Consult the manual page **more(C)** for the functionality.

11.4.13 Run

The Run Pipe Menu option allows the specification of any command not in the Pipe menu. The functionality is the same as the Visual Shell Main Menu Option "Run".

11.4.14 Sort

The XENIX sort utility can be invoked through the Sort Pipe menu option. The input stream is sorted.

The first Sort switch is *order*: < > . Select '>', the default, to sort in ascending order. Select '<' to sort in descending order.

The second Sort switch suppresses the case of characters in the sort. The default is off.

The third Sort switch sorts the input stream assuming an initial numeric field in the input stream. If this switch is off, initial numbers will be sorted in ascii order, which means that a line beginning with '10' will be output before the line beginning with '2'. The default is off.

The fourth Sort switch sorts the input stream in dictionary order, rather than ascii order.

The Sort Pipe Menu option corresponds to the XENIX command sort. Consult the manual page **sort(C)** for the functionality.

11.4.15 Tail

Tail prints a specified number of lines of the input stream up to the end of the stream. The *lines*: field may be set to specify the number of lines to print. The default is 15 lines.

The Tail Pipe Menu option corresponds to the XENIX command tail. Consult the manual page **tail(C)** for the functionality.

Appendix A

Ed

- A.1 Introduction A-1
- A.2 Demonstration A-1
- A.3 Basic Concepts A-1
 - A.3.1 The Editing Buffer A-2
 - A.3.2 Commands A-2
 - A.3.3 Line Numbers A-2
- A.4 Tasks A-2
 - A.4.1 Entering and Exiting The Editor A-2
 - A.4.2 Appending Text: a A-3
 - A.4.3 Writing Out a File: w A-4
 - A.4.4 Leaving The Editor: q A-4
 - A.4.5 Editing A New File: e A-5
 - A.4.6 Changing the File to Write Out to: f A-6
 - A.4.7 Reading in a File: r A-6
 - A.4.8 Displaying Lines On The Screen: p A-7
 - A.4.9 Displaying The Current Line: dot (.) A-9
 - A.4.10 Deleting Lines: d A-10
 - A.4.11 Performing Text Substitutions: s A-11
 - A.4.12 Searching A-13
 - A.4.13 Changing and Inserting Text: c and i A-16
 - A.4.14 Moving Lines: m A-18
 - A.4.15 Performing Global Commands: g and v A-19
 - A.4.16 Displaying Tabs and Control Characters: l A-21
 - A.4.17 Undoing Commands: u A-21
 - A.4.18 Marking Your Spot in a File: k A-22
 - A.4.19 Transferring Lines: t A-22
 - A.4.20 Escaping to the Shell: ! A-23
- A.5 Context and Regular Expressions A-23
 - A.5.1 Period: (.) A-24
 - A.5.2 Backslash: \ A-25
 - A.5.3 Dollar Sign: \$ A-27
 - A.5.4 Caret: ^ A-28
 - A.5.5 Star: * A-29
 - A.5.6 Brackets: [and] A-31
 - A.5.7 Ampersand: & A-32
 - A.5.8 Substituting New Lines A-33
 - A.5.9 Joining Lines A-34
 - A.5.10 Rearranging a Line: \ (and \) A-34
- A.6 Speeding Up Editing A-35
 - A.6.1 Semicolon: ; A-37
 - A.6.2 Interrupting the Editor A-38

- A.7 Cutting and Pasting with the Editor A-38
 - A.7.1 Inserting One File Into Another A-38
 - A.7.2 Writing Out Part of a File A-39
- A.8 Editing Scripts A-40
- A.9 Summary of Commands A-41

A.1 Introduction

Ed is a text editor used to create and modify text. The text is normally a document, a program, or data for a program, thus *ed* is a truly general purpose program. Note that the line editor *ex*, available with other XENIX packages is very similar to *ed*, and therefore this chapter can be used as an introduction to *ex* as well as to *ed*.

A.2 Demonstration

This section leads you through a simple session with *ed*, giving you a feel for how it is used and how it works. To begin the demonstration, invoke *ed* by typing:

```
ed
```

This invokes the editor and begins your editing session. An asterisk “*” prompts for commands to be entered. Initially, you are editing a temporary file that you can later copy to any file that you name. This temporary file is called the “editing buffer,” because it acts as a buffer between the text you enter and the file that you will eventually write out your changes to. Typically, the first thing you will want to do with an empty buffer is add text to it. For example, after the prompt, type:

```
a
this is line 1
this is line 2
this is line 3
this is line 4
CNTRL-D
```

This “appends” four lines of text to the buffer. To view these lines on your screen, type,

```
1,4p
```

where the “1,4” specifies a line number range and the **p** command “prints” the specified lines on the screen.

Now type

```
2p
```

to view line number two. Next type just

```
p
```

This prints out the current line on the screen, which happens to be line number two. By default, most *ed* commands operate on only the current line.

A.3 Basic Concepts

This section illustrates some of the basic concepts that you need to understand to effectively use *ed*.

A.3.1 The Editing Buffer

Each time you invoke *ed*, an area in the memory of the computer is allocated on which you will perform all of your editing operations. This area is called the "editing buffer". When you edit a file, the file is copied into this buffer where you will work on the copy of the original file. Only when you write out your file do you affect the original copy of the file.

A.3.2 Commands

Commands are entered by typing them at your keyboard. Like normal XENIX commands, entry of a command is ended by typing a NEWLINE. After you type NEWLINE the command is carried out. In the following examples, we will presume that entry of each command is completed by typing a NEWLINE, although this will not be explicitly shown in our examples. Most commands are single characters that can be preceded by the specification of a line number or a line number range. By default, most commands operate on the "current line", described below in the section on "Line Numbers". Many commands take filename or string arguments that are used by the command when it is executed.

A.3.3 Line Numbers

Any time you execute a command that changes the number of lines in the editing buffer, *ed* immediately renumbers the lines. At all times, every line in the editing buffer has a line number. Many editing commands will take either single line numbers or line number ranges as prefixing arguments. These arguments will normally specify the actual lines in the editing buffer that are to be affected by the given command. By default, a special line number called "dot" specifies the current line.

A.4 Tasks

This section discusses the tasks you perform in everyday editing. Frequently used and essential tasks are discussed near the beginning of this section. Seldom-used and special-purpose commands are discussed later.

A.4.1 Entering and Exiting The Editor

The simplest way to invoke *ed* is to type:

```
ed
```

The most common way, however, is to type:

```
ed filename
```

where *filename* is the name of a new or existing file.

To exit the editor, all you need to do is type:

```
q
```

If you have not yet written out the changes you have made to your file, *ed* warns you that you will

lose these changes by printing the message:

?

If you still want to quit, type another **q**. In most cases you will want to exit by typing:

w
q

so that you first write out your changes and only *then* exit the editor.

A.4.2 Appending Text: a

Suppose that you want to create some text starting from scratch. This section shows you how to put text in a file, just to get started. Later we'll talk about how to change it.

When you first invoke *ed*, it is like working with a blank piece of paper there is no text or information present. These must be supplied by the person using *ed*, usually by typing in the text, or by reading it in from a file. We will start by typing in some text and discuss how to read files later.

In *ed* terminology, the text being worked on is said to be "kept in a buffer". Think of the buffer as a workspace, or simply as a place where the information that you are going to be editing is kept. In effect, the buffer is the piece of paper on which you will write things, make changes, and finally file away.

You tell *ed* what to do to your text by typing instructions called "commands". Most commands consist of a single letter, each typed on a separate line. *Ed* prompts with an asterisk (*). This prompting can be turned on and off with the prompt command, **P**.

The first command we will discuss is append written as the letter "a" on a line by itself. It means "append (or add) text lines to the buffer, as they are typed in." Appending is like writing new material on a piece of paper.

To enter lines of text into the buffer, just type an "a", followed by a RETURN, followed by the lines of text you want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
```

To stop appending, type a line that contains only a period. The period (.) tells *ed* that you have finished appending. (You can also use CNTRL-D, but we will use the period throughout this discussion.) If *ed* seems to be ignoring you, type an extra line with just a period (.) on it. You may find you've added some garbage lines to your text, which you will have to take out later.

After appending is completed, the buffer contains the following three lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The **a** and **.** aren't there, because they are not text.

To add more text to what you already have, type another **a** command and continue typing your text.

If you make an error in the commands you type to *ed*, it will tell you by displaying the message:

```
?
error message
```

A.4.3 Writing Out a File: **w**

You will probably want to save your text for later use. To write out the contents of the buffer into a file, use the write command followed by the name of the file that you want to write to. This copies the contents of the buffer to the specified file, destroying any previous contents of the file. For example, to save the text in a file named *text*, type:

```
w text
```

Leave a space between **w** and the filename. *Ed* responds by printing the number of characters it has written out. For instance, *ed* might respond with

```
68
```

(Remember that blanks and the newline character at the end of each line are included in the character count.) Writing out a file just makes a copy of the text; the buffer's contents are not disturbed, so you can go on adding text to it. If you invoked *ed* with the command "*ed filename*", then by default a **w** command by itself will write the buffer out to *filename*.

This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. Writing out the text to a file from time to time as it is being created is a good idea. If the system crashes or if you make some horrible mistake, you will lose all the text in the buffer, but any text that was written out to a file is relatively safe.

A.4.4 Leaving The Editor: **q**

To terminate a session with *ed*, save the text you're working on by writing it to a file using the **w** command, then type:

```
q
```

The system responds with the XENIX prompt character. If you try to quit without writing out the file *ed* will print

?

At that point, write out the text if you want to save it; if not, typing another “q” will get you out of the editor.

Exercise

Enter *ed* and create some text by typing:

```
a
... text ...
.
```

Write it out by typing:

```
w filename
```

Then leave *ed* by typing:

```
q
```

Next, use the **cat** command to display the file on your terminal screen to see that everything has worked.

A.4.5 Editing A New File: e

A common way to get text into your editing buffer is to read it in from a file. This is what you do to edit text that you have saved with the **w** command in a previous session. The edit (**e**) command places the entire contents of a file in the buffer. If you had saved the three lines “Now is the time”, etc., with a **w** command in an earlier session, the *ed* command

```
e text
```

would place the entire contents of the file *text* into the buffer and respond with

```
68
```

which is the number of characters in *text*. *If anything is already in the buffer, it is deleted first.*

If you use the **e** command to read a file into the buffer, then you don’t need to use a filename after a subsequent **w** command. *Ed* remembers the last filename used in an **e** command, and **w** will write to this file. Thus, a good way to operate is this:

```
ed
e file
[editing session]
w
q
```

This way, you can type **w** from time to time and be secure in the knowledge that if you typed the filename right in the beginning, you are writing out to the proper file each time.

A.4.6 Changing the File to Write Out to: **f**

You can find out the last file written to at any time using the file command. Just type **f** without a filename. You can also change the name of the remembered filename with **f**. Thus a useful sequence is

```
ed precious
f junk
```

which gets a copy of the file named *precious*, then uses **f** to save the text in the file *junk*. The original file will be preserved as *precious*.

A.4.7 Reading in a File: **r**

Sometimes you want to read a file into the buffer without destroying what is already there. This function is useful for combining files. This is done with the read command. The command

```
r text
```

reads the file *text* into your editing buffer and adds it to the end of whatever is already in the buffer. For example, pretend that you have performed a read after an edit:

```
e text
r text
```

The buffer now contains *two* copies of *text* (i.e., six lines):

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the **w** and **e** commands, after the reading operation is complete **r** prints the number of characters read in.

Exercise

Experiment with the **e** command by reading and printing various files. You may get the error message

```
?name
cannot open input file
```

where *name* is the name of a nonexistent file. This means that the file doesn't exist, typically because you spelled the filename wrong, or perhaps because you do not have permission to read from or write to that file. Try alternately reading and appending to see how they work. Verify that the command

```
ed file.text
```


is equivalent to

```
ed
e file.text
```

A.4.8 Displaying Lines On The Screen: p

Use the “print” (**p**) command to print the contents of the editing buffer (or parts of it) on the terminal screen. Specify the lines where you want printing to begin and where you want it to end, separated by a comma and followed by the letter “p”. Thus, to print the first two lines of the buffer (that is, lines 1 through 2) type:

```
1,2p
```

Ed responds with:

```
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use “1,3p” as above if you knew there were exactly 3 lines in the buffer. But you will rarely know how many lines there are, so *ed* provides a shorthand symbol for the line number of the last line in the buffer: the dollar sign (\$). Use it this way:

```
1,$p
```

This will print *all* the lines in the buffer (from line 1 to the last line). If you want to stop the printing before it is finished, press the INTERRUPT key. *Ed* then displays

```
?
interrupt
```

and waits for the next command.

To print the *last* line of the buffer, use:

```
$p
```

You can print any single line by typing the line number, followed by a **p**. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number; there’s no need to type the letter **p**. If you type

\$

ed prints the last line of the buffer.

You can also use **\$** in combinations like

\$-1,\$p

which prints the last two lines of the buffer. This helps when you want to see how far you are in your typing.

The next step is to use address arithmetic to combine the line numbers like dot (.) and dollar sign (\$) with plus (+) and minus (-). (Note that "dot" is shorthand for the current line, and is discussed in a later section.) Thus

\$-1

prints the next to last line of the current file (that is, one line before the line \$). For example, to recall how far you were in a previous editing session

\$-5,\$p

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six lines in the file, you'll get an error message.

The command

.-3, +3p

prints from three lines before the current line (line dot) to three lines after. The plus (+) can be omitted:

.-3,3p

is identical in meaning.

Another area in which you can save typing effort in specifying lines is to use plus and minus as line numbers by themselves. For example

-

by itself is a command to move back one line in the file. In fact, you can string several minus signs together to move back that many lines. For example

moves back three lines, as does

-3

Thus

-3, +3p

is also identical to

```
?.-3p+3p
```

A.4.9 Displaying The Current Line: dot (.)

Suppose your editing buffer still contains the following six lines:

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

If you type

```
1,3p
```

ed displays

```
Now is the time
for all good men
to come to the aid of their party.
```

Try typing:

```
p
```

This prints

```
to come to the aid of their party.
```

which is the third line of the buffer. In fact, it is the last (most recent) line that you have done anything with. You can repeat this **p** command without line numbers, and *ed* will continue to print line 3.

This happens because *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. The line most recently acted on is referred to with a period (.) and is called “dot”. Dot is a line number in the same way that dollar (\$) is; it means “the current line”, or loosely, “the line you most recently did something to”. You can use it in several ways. One possibility is to type:

```
.,$p
```

This will print all the lines from (and including) the current line clear to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed. In the example above, **p** sets dot to 6.

Dot is often used in combinations like this one:

.+1

Or equivalently:

.+1p

This means “print the next line” and is one way of stepping slowly through the editing buffer. You can also type

.-1

This means “print the line *before* the current line”. This enables you to go backwards through the file if you wish. Another useful command is something like

.-3,.-1p

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing:

.=

Ed responds by printing the value of dot. Essentially, **p** can be preceded by zero, one, or two line numbers. If no line number is given, *ed* prints the “current line”, the line that dot refers to. If one line number is given (with or without the letter **p**), *ed* prints that line (and dot is set there); and if two line numbers are given, *ed* prints all the lines in that range (and sets dot to the last line printed). If two line numbers are specified, the first cannot be bigger than the second.

Pressing RETURN once causes printing of the next line. It is equivalent to:

.+1p

Try it. Next, try typing a minus sign (–) by itself; it is equivalent to typing

.-1p

Exercise

Create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print lines in reverse order using “3,1p” do not work.

A.4.10 Deleting Lines: **d**

Suppose you want to get rid of the three extra lines in the buffer. Use the delete command. Its action is similar to that of **p**, except that **d** deletes lines instead of printing them. The lines to be deleted are specified for **d** exactly as they are for **p**. Thus, the command

4,\$d

deletes lines 4 through the end. There are now three lines left in our example, as you can check by typing:

```
1,$p
```

Notice that **\$** now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **\$**.

Exercise

Experiment with the **a**, **e**, **r**, **w**, **p**, and **d** commands until you are sure that you know what they do, and until you understand how dot (**.**), dollar (**\$**), and line numbers are used.

Try using line numbers with **a**, **r**, and **w**, as well. You will find that **a** appends lines *after* the line number that you specify (rather than after dot); that **r** reads in a file *after* the line number you specify (not necessarily at the end of the buffer); and that **w** writes out exactly the lines you specify, not the whole buffer. These variations are sometimes useful. For instance, you can insert a file at the beginning of a buffer by typing

```
Or filename
```

and you can enter lines at the beginning of the buffer by typing:

```
0a
[input text here]
```

Notice that typing

```
.w
```

is very different from typing

```
.
w
```

since the former writes out only a single line and the latter writes out the whole file.

A.4.11 Performing Text Substitutions: **s**

One of the most important *ed* commands is the substitute command. This is the command that is used to change individual words or letters within a line or group of lines. It is the command used to correct spelling mistakes and typing errors.

Suppose that, due to a typing error, line 1 says:

```
Now is th time
```

The letter “e” has been left off of the word “the”. You can use **s** to fix this up as follows:

```
1s/th/the/
```

This substitutes for the characters “th”, the characters “the”, in line 1. To verify that the substitution has worked, type

```
p
```

to get

```
Now is the time
```

which is what you wanted. Notice that dot must be the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The syntax for the substitute command follows:

```
[starting-line,ending-line]s/pattern/replacement/ cmds
```

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. Changing *every* occurrence is discussed later in this section. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (If no substitution takes place, dot is *not* changed. This causes printing of the error message:

```
?  
search string not found
```

Thus, you can type

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text.

If no line numbers are given, the **s** command assumes we mean “make the substitution on line dot”, so it changes things only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes a correction on the current line, then prints it to make sure the correction worked out right. If it didn't, you can try again. (Notice that the **p** is on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multicommand lines are legal.)

It is also legal to type

```
s/string//
```

which means “change the first string of characters to nothing” or, in other words, remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

```
Nowxx is the time
```

you could type


```
s/xx//p
```

to get

Now is the time

Notice that two adjacent slashes mean “no characters”, not a space. There *is* a difference.

Exercise

Experiment with the substitute command. See what happens if you substitute a word on a line with several occurrences of that word. For example, type:

```
a
the other side of the coin
.
s/the/on the/p
```

This results in:

on the other side of the coin

A substitute command changes only the *first* occurrence of the first string. You can change all occurrences by adding a **g** (for “global”) to the **s** command, like this:

```
s/ ... / ... /g
```

Try using characters other than slashes to delimit the two sets of characters in the **s** command—anything should work except spaces or tabs.

A.4.12 Searching

Now that you’ve mastered the substitute command, you can move on to mastering another important concept: context searching.

Suppose you have the original three-line text in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

Suppose you want to find the line that contains the word “their”, so that you can change it to the word “the”. With only three lines in the buffer, it’s pretty easy to keep track of which line the word “their” is on. But if the buffer contained several hundred lines, and you’d been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of its number, by specifying a textual pattern contained in the line.

The way to say “search for a line that contains this particular string of characters” is to type:

```
/string of characters we want to find/
```

For example, the *ed* command

```
/their/
```

is a context search sufficient to find the desired line it will locate the next occurrence of the characters between the slashes (i.e., “their”). Note that you do not need to type the final slash. The above search command is the same as typing:

```
/their
```

The search command sets dot to the line on which the pattern is found and prints it for verification:

```
to come to the aid of their party.
```

“Next occurrence” means that *ed* starts looking for the string at line “.+1”, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search “wraps around” from \$ to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot. If the given string of characters can't be found in any line, *ed* prints the error message:

```
?  
search string not found
```

Otherwise, *ed* prints the line it found. You can also search *backwards* in a file for search strings by using question marks instead of slashes. For example

```
?thing?
```

searches backwards in the file for the word “thing” as does

```
?thing
```

This is especially handy when you realize that the string you want is backwards from the current line.

The slash and question mark are the only characters you can use to delimit a context search, though you can use any character in a substitute command. If you get unexpected results using any of the characters

```
^ . $ [ * \ &
```

read Section A.5, “Context and Regular Expressions”.

You can do both the search for the desired line *and* a substitution at the same time, like this:

```
/their/s/their/the/p
```

This yields:

```
to come to the aid of the party.
```

The above command contains three separate actions. The first is a context search for the desired line, the second is the substitution, and the third is the printing of the line.

The expression “/their/” is a context search expression. In their simplest form, all context search expressions are like this a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like `s`. They were used both ways in the previous examples.

Suppose the buffer contains the three familiar lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The *ed* line numbers

```
/Now/+1
/good/
/party/-1
```

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could type

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

The choice is dictated only by convenience. For instance, you could print all three lines by typing

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

or any similar combination. The first combination is better if you don't know how many lines are involved.

The basic rule is that a context search expression is the same as a line number, so it can be used wherever a line number is needed.

Suppose you search for

```
/horrible thing/
```

and when the line is printed you discover that it isn't the “horrible thing” that you wanted, so it is necessary to repeat the search. You don't have to retype the search, because the construction

```
//
```

is a shorthand expression for “the previous thing that was searched for”, whatever it was. This can be repeated as many times as necessary. You can also go backwards, since

```
??
```

searches for the same thing, but in the reverse direction.

You can also use `//` as the left side of a substitute command, to mean “the most recent pattern”. For example, examine:

```
/horrible thing/
```

Ed prints the line containing “horrible thing”.

```
s//good/p
```

This changes “horrible thing” to “good”. To go backwards and change “horrible thing” to “good”, type:

```
??s//good/
```

Exercise

Experiment with context searching. Scan through a body of text with several occurrences of the same string of characters using the same context search.

Try using context searches as line numbers for the substitute, print, and delete commands. (Context searches can also be used with the `r`, `w`, and `a` commands.)

Try context searching using instead of This scans lines in the buffer in reverse order instead of normal order, which is sometimes useful if you go too far while looking for a string of characters. It's an easy way to back up in the file you're editing.

If you get unexpected results with any of the characters

```
^ . $ [ * \ &
```

read Section A.4, “Context and Regular Expressions”.

A.4.13 Changing and Inserting Text: `c` and `i`

This section discusses the change command, which is used to change or replace one or more lines, and the insert command, which is used for inserting one or more lines.

The `c` command is used to replace a number of lines with different lines that you type at the terminal. For example, to change lines “`.+1`” through “`$`” to something else, type:

```
.+1,$c  
type the lines of text you want here ...  
.
```


The lines you type between the **c** command and the dot (.) will replace the originally addressed lines. This is useful in replacing a line or several lines that have errors in them.

If only one line is specified in the **c** command, then only that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of a period to end the input. This works just like the period in the append command and must appear by itself on a new line. If no line number is given, the current line specified by dot is replaced. The value of dot is set to the last line you typed in. Note that the terminating period and the line referenced by dot are completely different: the first is used simply to terminate a command, the second points at a specific line of text.

The **i** command is similar to the append command. For example

```
/string/i
type the lines to be inserted here ...
.
```

inserts the given text *before* the next line that contains “string”. The text between **i** and the terminating period is *inserted before* the specified line. If no line number is specified, dot is used. Dot is set to the last line inserted.

Exercise

The **c** command is like a combination of delete followed by insert. Experiment to verify that

```
start,end d
i
[text]
.
```

is almost the same as

```
start,end c
[text]
.
```

These are not *precisely* the same if the last line gets deleted.

Experiment with **a** and **i** to see that they are similar, but not the same. Observe that

```
line-number a
[text]
.
```

appends *after* the given line, while

```
line-number i
[text]
.
```

inserts *before* it. If no line number is given, **i** inserts before line dot, while **a** appends after line dot.

A.4.14 Moving Lines: m

The move command lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You *could* do it by typing

```
1,3w temp
$r temp
1,3d
```

where *temp* is the name of a temporary file. However, you can do it more easily with the **m** command:

```
1,3m$
```

This will move lines 1 through 3 to the end of the file.

The general case is

```
start-line,end-linemafter-this-line
```

There is a third line to be specified: the place where the moved text gets put. Of course, the lines to be moved can be specified by context searches. If you had

```
First paragraph
end of first paragraph.
Second paragraph
end of second paragraph.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/-1
```

Notice the **-1**. The moved text goes *after* the line mentioned. Dot gets set to the last line moved. Your file will now look like this:

```
Second paragraph
end of second paragraph
First paragraph
end of first paragraph
```

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first line after the second line. Suppose that you are positioned at the first line. Then

```
m+
```

moves line dot to one line after the current line dot. If you are positioned on the second line,

```
m--
```

moves line dot to one line after the current line dot.

The **m** command is more succinct than writing, deleting and rereading. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target,

you have to take care to specify them properly, or you may not move the lines you want. The result of a bad **m** command can be a mess. Doing the job one step at a time makes it easier for you to verify at each step that you accomplished what you wanted. It is also a good idea to issue a **w** command before doing anything complicated; then if you make a mistake, it's easy to back up to where you were.

For more information on moving text, see Section A.4.18, “Marking Your Spot in a File:k”.

A.4.15 Performing Global Commands: **g** and **v**

The “global” commands **g** and **v** are used to execute one or more editing commands on all lines that either contain or don't contain a specified pattern.

For example, the command

```
g/XENIX/p
```

prints all lines that contain the word “XENIX”. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

For example,

```
g/^./p
```

prints all the *troff* formatting commands in a file (lines that begin with “.”). (For an explanation of the use of the caret (^) and the backslash (\) see Section A.5, “Context and Regular Expressions”.

The **v** command is identical to **g**, except that it operates on those lines that do *not* contain an occurrence of the pattern. (Mnemonically, the “v” can be thought of as part of the word

For example

```
v/^./p
```

prints all the lines that don't begin with a period (i.e., the actual text lines).

Any command can follow **g** or **v**. For example, the following command deletes all lines that begin with “.”:

```
g/^./d
```

This command deletes all empty lines:

```
g/^$/d
```

Probably the most useful command that can follow a global command is the substitute command. For example, we could change the word “Xenix” to “XENIX” everywhere, and verify that it really worked, with

```
g/Xenix/s//XENIX/gp
```

Notice that we used `//` in the substitute command to mean “the previous pattern”, in this case, “Xenix”. The `p` command executes on each line that matches the pattern, not just on those in which a substitution took place.

The global command makes two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line is examined in turn, dot is set to that line, and the command executed. This means that it is possible for the command that follows a `g` or `v` command to use addresses, set dot, and so on, quite freely. For example:

```
g/^\.P/+
```

prints the line that follows each “.P” command (the signal for a new paragraph in some formatting packages). Remember that plus (+) means “one line past dot”. And

```
g/topic/?^\.H?p
```

searches for each line that contains the word “topic”, scans backwards until it finds a line that begins with a “.H” (a heading) and prints it, thus showing the headings under which “topic” is mentioned. Finally

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with “.EQ” and “.EN” formatting commands.

The `g` and `v` commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

It is possible to give more than one command under the control of a global command. For example, suppose the task is to change “x” to “y” and “a” to “b” on all lines that contain “thing”. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The backslash (\) signals the `g` command that the set of commands continues on the next line; the `g` command terminates on the first line that does not end with a backslash.

Note that you cannot use a substitute command to insert a new line within a `g` command. Watch out for this.

The command

```
g/x/s//y/\
s/a/b/
```

does *not* work as you might expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be “x” (as expected), and sometimes it will be “a” (not expected). You must spell it out, like this:


```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands as part of a global command. As with other multiline constructions, add a backslash at the end of each line except the last. Thus, to add an “.nf” and “.sp” command before each “.EQ” line, type:

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a period (.) to terminate the **i** command, unless there are further commands to be executed under the global command.

A.4.16 Displaying Tabs and Control Characters: **l**

Ed provides two commands for printing the contents of the text you are editing. You should already be familiar with **p**, in combinations like

```
l,$p
```

to print all the lines you are editing, or

```
s/abc/def/p
```

to change “abc” to “def” on the current line. Less familiar is the “list” (**l**) command which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** prints each tab as “>” and each backspace as “<”. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also “folds” long lines for printing. Any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash (\), so you can tell it was folded. This is useful for printing lines longer than the width of your terminal screen.

Occasionally, the **l** command will print a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don’t print, like form feed, vertical tab, or bell. Each backslash-number combination represents a single ASCII character. Note that numbers are octal and not decimal. When you see such characters, be wary: they may have surprising meanings when printed on some terminals. Often their presence indicates an error in typing, because they are rarely used.

A.4.17 Undoing Commands: **u**

Occasionally you will make a substitution in a line, only to realize too late that it was a mistake. The undo command, lets you “undo” the last substitution. Thus the last line that was substituted can be restored to its previous state by typing:

```
u
```

This command does *not* work with the **g** and **v** commands.

A.4.18 Marking Your Spot in a File: **k**

The mark command, **k**, provides a facility for marking a line with a particular name, so that you can later reference it by name, regardless of its actual line number. This can be handy for moving lines and keeping track of them as they move. For example

```
kx
```

marks the current line with the name "x". If a line number precedes the **k**, that line is marked. (The mark name must be a single lowercase letter.) You can refer to the marked line with the notation:

```
'x
```

Note the use of the single quotation mark (') here. Marks are very useful for moving things around. Find the first line of the block to be moved and then mark it with:

```
ka
```

Then find the last line and mark it with

```
kb
```

Go to at the place where the text is to be inserted and type:

```
'a,'bm.
```

A line can have only one mark name associated with it at any given time.

A.4.19 Transferring Lines: **t**

We mentioned earlier the idea of saving lines that are hard to type or used often, to cut down on typing time. Ed provides another command, called **t** (for transfer) for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing.

A common use for **t** is to create a series of lines that differ only slightly. For example, you can type


```

a
Now is the time for all good men to come to the aid of their party.
.
t.                                [make a copy]
s/men/women/                     [change it a bit]
t.                                [make third copy]
s/Now is/yesterday was/ [change it a bit]

```

Your file will look like this:

```

Now is the time for all good men to come to the aid of their party.
Now is the time for all good women to come to the aid of their party.
Yesterday was the time for all good women to come to the aid of their party.

```

A.4.20 Escaping to the Shell: !

Sometimes it is convenient to temporarily escape from the editor to execute a XENIX command without leaving the editor. The shell escape command, provides a way to do this.

If you type

```
!command
```

your current editing state is suspended, and the XENIX command you asked for is executed. When the command finishes, *ed* will signal you by printing another exclamation (!); at that point you can resume editing.

A.5 Context and Regular Expressions

You may have noticed that things don't work right when you use characters such as the period (.), the asterisk (*), and the dollar sign (\$) in context searches and with the substitute command. The reason is rather complex, although the solution to the problem is simple. Ed treats these characters as special. For instance, in a context search or the first string of the substitute command, the period (.) means "any character", not a period, so

```
/x.y/
```

means a line with an "x", any character, and a "y", not just a line with an "x", a period, and a "y". A complete list of the special characters that can cause trouble follows:

```
^ . $ [ * \ /
```

The next few subsections discuss how to use these characters to describe patterns of text in search and substitute commands. These patterns are called "regular expressions", and occur in several other important XENIX commands and utilities, including (See the *XENIX Reference Manual*).

Recall that a trailing g after a substitute command causes all occurrences to be changed. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first command replaces the *first* “this” on the line with “that”. If there is more than one “this” on the line, the second form with the trailing **g** changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to print or list the contents of the line. For example, all of the following are legal and mean slightly different things:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two line numbers to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of “mispell” to “misspell” in each line of the file. But

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in each line (and this is more likely to be what you wanted).

If you add a **p** or **l** to the end of any of these substitute commands, only the last line changed is printed, not all the lines. We will talk later about how to print all the lines that were modified.

A.5.1 Period: (.)

The first metacharacter that we will discuss is the period (**.**). On the left side of a substitute command, or in a search, a period stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where “x” and “y” occur separated by a single character, as in

```
x+y
x-y
x y
xzy
```

and so on.

Since a period matches a single character, it gives you a way to deal with funny characters printed by **l**. Suppose you have a line that appears as

```
th\07is
```

when printed with the **l** command, and that you want to get rid of the **\07**, which represents an AS-

CII bell character.

The most obvious solution is to try

```
s/\07//
```

but this will fail. Another solution is to retype the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big. But for a very long line, retyping is not the best solution. This is where the metacharacter "." comes in handy. Since \07 really represents a single character, if we type

```
s/th.is/this/
```

the job is done. The period matches the mysterious character between the "h" and the "i", whatever it is.

Since the period matches any single character, the command

```
s/./,/
```

converts the first character on a line into a comma (,), which very often is not what you intended. The special meaning of the period can be removed by preceding it with a backslash.

As is true of many characters in ed, the period (.) has several meanings, depending on its context. This line shows all three:

```
.s/././
```

The first period is the line number of the line we are editing, which is called "dot". The second period is a metacharacter that matches any single character on that line. The third period is the only one that really is an honest, literal period. (Remember that a period is also used to terminate input from the **a** and **i** commands.) On the *right* side of a substitution, the period (.) is not special. If you apply this command to the line

```
Now is the time.
```

the result is

```
.ow is the time.
```

which is probably not what you intended. To change the period at the end of the sentence to a comma, type

```
s/\./,/
```

The special meaning of the period can be removed by preceding it with a backslash.

A.5.2 Backslash: \

Since a period means "any character", the question naturally arises: what do you do when you really want a period? For example, how do you convert the line

Now is the time.

into

Now is the time?

The backslash (\) turns off any special meaning that the next character might have; in particular, “\.” converts the “.” from a “match anything” into a literal period, so you can use it to replace the period in “Now is the time.” like this:

```
s/\./?/
```

The pair of characters “\.” is considered by ed to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.DE
```

at the start of a line. The search

```
/.DE/
```

isn't adequate, for it will find lines like

```
JADE  
FADE  
MADE
```

because the “.” matches the letter “A” on each of the lines in question. But if you type

```
/\.
```

only lines that contain “.DE” are found.

The backslash can be used to turn off special meanings for characters other than the period. For example, consider finding a line that contains a backslash. The search

```
/\
```

won't work, because the backslash (\) isn't a literal backslash, but instead means that the second slash (/) no longer delimits the search. By preceding a backslash with another backslash, you can search for a literal backslash:

```
/\\
```

You can search for a forward slash (/) with

```
/\//
```

The backslash turns off the special meaning of the slash immediately following so that it doesn't terminate the slash-slash construction prematurely.

A miscellaneous note about backslashes and special characters: you can use any character to del-

limit the pieces of an `s` command; there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains several slashes already, such as

```
//exec //sys.fort.go // etc...
```

you could use a colon as the delimiter. To delete all the slashes, type

```
s:/::g
```

The result is:

```
exec sys.fort.go etc...
```

When you are adding text with `a` or `i` or `c`, the backslash has no special meaning, and you should only put in one backslash for each one you want.

Exercise

Find two substitute commands, each of which converts the line

```
\x\.\y
```

into the line

```
\x\y
```

Here are several solutions; you should verify that each works:

```
s/\\\.//
s/x../x/
s/..y/y/
```

A.5.3 Dollar Sign: \$

The dollar sign “\$”, stands for “the end of the line”. Suppose you have the line

```
Now is the
```

and you want to add the word “time” to the end. Use the dollar sign (\$) like this:

```
s/$/ time/
```

to get

```
Now is the time
```

A space is needed before “time” in the substitute command, or you will get:

Now is the time

You can replace the second comma in the following line with a period without altering the first.

Now is the time, for all good men,

The command needed is:

```
s/,$/./
```

to get

Now is the time, for all good men.

The dollar sign (\$) here provides context to make specific which comma we mean. Without it the s command would operate on the first comma to produce:

Now is the time. for all good men,

To convert:

Now is the time.

into

Now is the time?

as we did earlier, we can use:

```
s/.$/?/
```

Like the period (.), the dollar sign (\$) has multiple meanings depending on context. In the following line

```
$s/$/$/
```

the first "\$" refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign to be added to that line.

A.5.4 Caret: ^

The caret (^) stands for the beginning of the line. For example, suppose you are looking for a line that begins with "the". If you simply type

```
/the/
```

you will probably find several lines that contain "the" in the middle before arriving at the one you want. But with


```
/^the/
```

you narrow the context, and thus arrive at the desired line more easily.

The other use of the caret (^) enables you to insert something at the beginning of a line. For example

```
s/^/ /
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

```
.P
```

you can use the command

```
/^\.P$/
```

A.5.5 Star: *

Suppose you have a line that looks like this:

```
text x   y text
```

where “text” stands for lots of text, and there are an indeterminate number of spaces between the “x” and the “y”. Suppose the job is to replace all the spaces between “x” and “y” with a single space. The line is too long to retype, and there are too many spaces to count.

This is where the metacharacter “star” (*) comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, type:

```
s/x *y/x y/
```

The “*” means “as many spaces as possible”. Thus “x *y” means an “x”, as many spaces as possible, then a “y”.

The star can be used with any character, not just a space. If the original example was

```
text x-----y text
```

then all minus signs (–) can be replaced by a single space with the command:

```
s/x-*y/x y/
```

Finally, suppose that the line was:

```
text x.....y text
```

If you blindly type

```
s/x.*y/x y/
```

The result is unpredictable. If there are no other x's or y's on the line, the substitution will work, but not necessarily. The period matches *any* single character so the “.” matches as many single characters as possible, and unless you are careful, it can remove more of the line than you expected. For example, if the line was like this

```
x text x.....y text y
```

then typing

```
s/x.*y/x y/
```

takes everything from the *first* “x” to the *last* “y”, which, in this example, is undoubtedly more than you wanted.

The solution is to turn off the special meaning of the period (.) with the backslash (\):

```
s/x\.*y/x y/
```

Now the substitution works, for “\.” means “as many periods as possible”.

There are times when the pattern “.” is exactly what you want. For example, to change

```
Now is the time for all good men ....
```

into

```
Now is the time.
```

use “.” to remove everything after the “for”:

```
s/for.*./
```

There are a couple of additional pitfalls associated with the star (*). Most notable is the fact that “as many as possible” means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

```
xy□text□x□□y□text
```

where the squares represent spaces, and we said

```
s/x□*y/x□y/
```

the first “xy” matches this pattern, for it consists of an “x”, zero spaces, and a “y”. The result is that the substitute acts on the first “xy”, and does not touch the later one that actually contains some intervening spaces.

The way around this is to specify a pattern like

```
/x□□*y/
```

which says an “x”, a space, then as many more spaces as possible, then a “y”, in other words, one

or more spaces.

The other pitfall associated with the star (*) again relates to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this is that zero is a legitimate number of matches, and there are no x's at the beginning of the line (so that gets converted into a "y"), nor between the "a" and the "b" (so that gets converted into a "y"), and so on. If you don't want zero matches, use

```
s/xx*/y/g
```

since "xx*" is one or more x's.

A.5.6 Brackets: [and]

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might try a series of commands like

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all the numbers are gone, you must get all the digits on one pass. That is the purpose of the brackets.

The construction

```
[0123456789]
```

matches any single digit the whole thing is called a "character class". With a character class, the job is easy. The pattern "[0123456789]*" matches zero or more digits (an entire number), so

```
1,$s/^ [0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and there are only three special characters (^,], and -) inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can type:

```
/[\.$^[]/
```

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0–9]; similarly, [a–z] stands for the lowercase letters, and [A–Z] for uppercase.

Within [], the “[” is not special. To get a “[” (or a “–”) into a character class, make it the first character.

You can also specify a class that means “none of the following characters”. This is done by beginning the class with a caret (^). For example

```
[^0-9]
```

stands for “any character *except* a digit”. Thus, you might find the first line that doesn't begin with a tab or space with a search like:

```
/^[^(space)(tab)]/
```

Within a character class, the caret has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^]/
```

finds a line that doesn't begin with a caret.

A.5.7 Ampersand: &

To save typing, the ampersand (&) can be used in substitutions to signify the string of text that was found on the left side of a substitute command. Suppose you have the line

```
Now is the time
```

and you want to make it:

```
Now is the best time
```

You can type:

```
s/the/the best/
```

It's unnecessary to repeat the word “the”. The ampersand (&) eliminates this repetition. On the *right* side of a substitution, the ampersand means “whatever was just matched”, so you can type

```
s/the/& best/
```

and the ampersand will stand for “the”. This isn't much of a saving if the thing matched is just “the”, but if the match is very long, or if it is something like “.*” which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to put parentheses in a line, regardless of its length, type:


```
s/.*/(&)/
```

The ampersand can occur more than once on the right side. For example

```
s/the/ & best and & worst/
```

makes

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand use the backslash to turn off the special meaning. For example

```
s/ampersand/\&/
```

converts the word into the symbol. The ampersand is not special on the left side of a substitute command, only on the right side.

A.5.8 Substituting New Lines

Ed provides a facility for splitting a single line into two or more shorter lines by “substituting in a newline”. For example, suppose a line has become unmanageably long because of editing. If it looks like

```
text xy text
```

you can break it between the “x” and the “y” like this:

```
s/xy/x\  
y/
```

This is actually a single command, although it is typed on two lines. Because the backslash (\) turns off special meanings, a backslash at the end of a line makes the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As an example, consider italicizing the word “very” in a long line by splitting “very” onto a separate line, and preceding it with the formatting command “.I”. Assume the line in question looks like this:

```
text a very big text
```

The command

```
s/ very /\n.I\/
```

converts the line into four shorter lines, preceding the word “very” with the line “.I”, and eliminating the spaces around the “very” at the same time.

When a new line is substituted in a string, dot is left at the last line created.

A.5.9 Joining Lines

Lines may be joined together, with the **j** command. Assume that you are given the lines:

```
Now is  
the time
```

Suppose that dot is set to the first line. Then the command

```
j
```

joins them together to produce:

```
Now is the time
```

No blanks are added, which is why a blank was shown at the beginning of the second line.

All by itself, a **j** command joins the lines signified by dot and dot~+~1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines in a file into one big line and prints it.

A.5.10 Rearranging a Line: \ (and \)

Recall that “&” is shorthand for whatever was matched by the left side of an **s** command. In much the same way, you can capture separate pieces of what was matched. The only difference is that you have to specify on the left side just what pieces you’re interested in.

Suppose that you have a file of lines that consist of names in the form

```
Smith, A. B.  
Jones, C.
```

and so on, and you want the initials to precede the name, as in:

```
A. B. Smith  
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone.

The alternative is to “tag” the pieces of the pattern (in this case, the last name, and the initials), then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between `\(` and `\)`, whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol, “`\1`”, refers to whatever matched the first `\(...\)` pair; “`\2`”, to the second `\(...\)`, and so on.

The command

```
1,$s/^\([.*]\), *\(.*\) / \2 \1 /
```

although hard to read, does the job. The first `\(...\)` matches the last name, which is any string up to the comma; this is referred to on the right side with “`\1`”. The second `\(...\)` is whatever follows the comma and any spaces, and is referred to as “`\2`”.

With any editing sequence this complicated, it’s unwise to simply run it and hope. The global commands **g** and **v** provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

A.6 Speeding Up Editing

One of the most effective ways to speed up your editing is knowing what lines will be affected by a command if you don’t specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

For example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains “thing”. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there is no occurrence of “thing”? Dot is unchanged. This is also true if the cursor was on the only occurrence of “thing” when you issued the command. The same rules hold for searches that use `?...?`; the only difference is the direction in which you search.

The delete command, **d**, leaves dot pointing at the line that followed the last deleted line. When the line dollar (\$) gets deleted, however, dot points at the *new* line \$.

The line-changing commands **a**, **c**, and **i**, by default, all affect the current line. If you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

The **a**, **c**, and **i** commands behave identically in one respect when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want when typing and edi-

ting on the fly. For example, you can type

```
a
text
botch (minor error)
.
s/botch/correct/ (fix botched line)
a
more text
.
```

without specifying any line number for the substitute command or for the second append command. Or you can type:

```
a
text
horrible botch (major error)
.
c      (replace entire line)
fixed up line
.
```

Experiment to determine what happens if you add *no* lines with an **a**, **c**, or **i** command.

The **r** command reads a file into the text being edited, at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even type

Or

to read a file in at the beginning of the text. (You can also type *0a* or *li* to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written out. If you precede it by two line numbers, that range of lines is written out. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written out. This is true even if you type something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

(Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you accidentally delete what you're editing.)

The general rule is simple: you are left sitting on the last line changed; if there were no changes, then dot is unchanged. To illustrate, suppose that there are three lines in the buffer, and the line given by dot is the middle one:

```
x1
x2
x3
```


Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, only the first line would be changed and printed, and that is where dot would be set.

A.6.1 Semicolon;

Searches with `/.../` and `?...?` start at the current line and move forward or backward, respectively, until they either find the pattern or get back to the current line. Sometimes this is not what you want. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
.
```

Starting at line 1, you would expect the command

```
/a/,/b/p
```

to print all the lines from the “ab” to the “bc” inclusive. This is not what happens. *Both* searches (for “a” and for “b”) start from the same point, and thus they both find the line that contains “ab”. As a result, a single line is printed. Worse, if there had been a line with a “b” in it before the “ab” line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn’t set dot as each address is processed; each search starts from the same place. In ed, the semicolon (;) can be used just like the comma, with the single difference that use of a semicolon forces dot to be set at the time the semicolon is encountered, as the line numbers are being evaluated. In effect, the semicolon “moves” dot. Thus, in our example above, the command

```
/a;/b/p
```

prints the range of lines from “ab” to “bc”, because after the “a” is found, dot is set to that line, and then “b” is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second*

occurrence of “thing”. You could type

```
/thing/  
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to type:

```
/thing/; //
```

This says “find the first occurrence of “thing”, set dot to that line, then find the second occurrence and print only that”.

Closely related is searching for the second to last occurrence of something, as in:

```
?something?;??
```

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to type

```
1;/thing/
```

because if “thing” occurs on line 1 it won't be found. The command

```
0;/thing/
```

will work because it starts the search at line 1. This is one of the few places where 0 is a legal line number.

A.6.2 Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you press the INTERRUPT key while ed is executing a command, your file is restored, as much as possible, to what it was before the command began. Naturally, some changes are irrevocable if you are reading in or writing out a file, making substitutions, or deleting lines. These will be stopped in some unpredictable state in the middle (which is why it usually unwise to stop them). Dot may or may not be changed.

If you are using the print command, dot is not changed until the printing is done. Thus, if you decide to print until you see an interesting line, and then press INTERRUPT, to stop the command, dot will not *not* be set to that line or even near it. Dot is left where it was when the **p** command was started.

A.7 Cutting and Pasting with the Editor

This section describes how to manipulate pieces of files, individual lines or groups of lines.

A.7.1 Inserting One File Into Another

Suppose you have a file called *memo*, and you want the file called *table* to be inserted just after a reference to Table 1. That is, in *memo* somewhere is a line that says

Table 1 shows that ...

and the data contained in *table* has to go there.

To put *table* into the correct place in the file edit *memo*, find “Table 1”, and add the file *table* right there:

```
ed memo
/Table 1/
response from ed
.r table
```

The critical line is the last one. The **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command, without any address, adds lines at the end, so it is the same as “**\$r**”.

A.7.2 Writing Out Part of a File

The other side of the coin is writing out part of the document you’re editing. For example, you may want to split the table from the previous example out into a separate file so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
[lots of stuff]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called *table*, first find the start of the table (the “.TS” line), then write out the interesting part. For example, first type:

```
/^\.TS/
```

This prints out the found line:

```
.TS
```

Next type

```
./^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. If you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed

later, then save it don't retype it. For example, in the editor, type:

```
a
lots of stuff
horrible line
.
.w temp
a
more stuff
.
.r temp
a
more stuff
.
```

A.8 Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a "script", i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every "Xenix" to "XENIX" and every "USA" to "America" in a large number of files. Put the following lines into the file *script*:

```
g/Xenix/s//XENIX/g
g/USA/s//America/g
w
q
```

Now you can type:

```
ed - file1 <script
ed - file2 <script
...
```

This causes *ed* to take its commands from the prepared file *script*. Notice that the whole job has to be planned in advance, and that by using the XENIX shell command interpreter, you can cycle through a set of files automatically. The dash (-) suppresses unwanted messages from *ed*.

When preparing editing scripts, you may need to place a period as the only character on a line to indicate termination of input from an *a* or *i* command. This is difficult to do in *ed*, because the period you type will terminate input rather than be inserted in the file. Using a backslash to escape the period won't work either. One solution is to create the script using a character such as the at-sign (@) to indicate end of input. Then, later, use the following command to replace the at-sign with a period:

```
s/^@$/./
```


A.9 Summary of Commands

The following is a list of all ed commands. The general form of ed commands is the command name, preceded by one or two optional line numbers and, in the case of **e**, **f**, **r**, and **w**, followed by a filename. Only one command is allowed per line, but a **p** command may follow any other command (except **e**, **f**, **r**, **w**, and **q**).

- a** Appends, i.e., adds lines to the buffer (at line dot, unless a different line is specified). Appending continues until a period is typed on a new line. The value of dot is set to the last line appended.
- c** Changes the specified lines to the new text which follows. The new lines are terminated by a period on a new line, as with **a**. If no lines are specified, replace line dot. Dot is set to the last line changed.
- d** Deletes the lines specified. If none are specified, deletes line dot. Dot is set to the first undeleted line following the deleted lines unless dollar (\$) is deleted, in which case dot is set to dollar.
- e** Edits a new file. Any previous contents of the buffer are thrown away, so issue a **w** command first.
- f** Prints the remembered filename. If a name follows **f**, then the remembered name is set to it.
- g** The command *g/string/commands* executes *commands* on those lines that contain *string*, which can be any context search expression.
- i** Inserts lines before specified line (or dot) until a single period is typed on a new line. Dot is set to the last line inserted.
- l** Lists lines, making visible nonprinting ASCII characters and tabs. Otherwise similar to **p**.
- m** Moves lines specified to after the line named after **m**. Dot is set to the last line moved.
- p** Prints specified lines. If none are specified, print the line specified by dot. A single line number is equivalent to a command. A single RETURN prints “.+1”, the next line.
- q** Quits ed. Your work is not saved unless you first give a **w** command. Give it twice in a row to abort edit.
- r** Reads a file into buffer (at end unless specified elsewhere.) Dot is set to the last line read.
- s** The command “*s/string1/string2/*” substitutes the pattern matched by *string1* with the string specified by *string2* in the specified lines. If no lines are specified, the substitution takes place only on the line specified by dot. Dot is set to the last line in which a substitution took place, which means that if no substitution takes place, dot remains unchanged. The **s** command changes only the first occurrence of *string1* on a line; to change multiple occurrences on a line, type a **g** after the final slash.

- t** Transfers specified lines to the line named after **t**. Dot is set to the last line moved.
- v** The command *v/string/commands* executes commands on those lines that do not contain *string*.
- u** Undoes the last substitute command.
- w** Writes out the editing buffer to a file. Dot remains unchanged.
- .=** Prints value of dot. (An equal sign by itself prints the value of \$.)

!command

The line *!cmd-line* causes *cmd-line* to be executed as a XENIX command.

/string/

Context search. Searches for next line which contains this string of characters and prints it. Dot is set to the line where string was found. The search starts at *.+1*, wraps around from *\$* to 1, and continues to dot, if necessary.

?string?

Context search in reverse direction. Starts search at *.-1*, scans to 1, wraps around to *\$*.

Index

{ } command See Braces command ({})
: command See Colon command (:)
. command See Dot command (.)
! command See escape command (!)
/ command See Vi
. command
 vi 5-3
 vi use See Vi
* See Asterisk (*)
[] See Brackets ([])
- See Dash (-)
)
 . See Period (.)
 ? See Question mark (?)
 / See Slash (/)
 \$# variable, argument recording 7-9
 \$? variable, command exit status
 7-9
 \$- variable, execution flags 7-10
 \$! variable, process number 7-10
0 command See Vi

A

a character, permission change 4-13
a command
 appending See Ed
 ed use See Ed
 mail 6-15
 mail 6-25
 mail 6-9
-a operator 7-25
-a option
 function 3-7
Absolute pathname See Pathname
Account, new user 2-1
Addition See BC
Addition See Calculation
Alias command See a command
Alias
 C-shell use See C-shell
Alphabetizing See sort command
Ampersand (&)
 and-if operator symbol See And-if
 operator (&&)
 background command 3-7
 background process 4-17
 background process 7-14
 background process 7-40
 command list 7-13
 ed use See Ed
 interrupt, quit immunity 7-14
 jobs to other computers 7-14
 metacharacter See Ed
 off-line printing 7-14
 use restraint 7-14
And-if operator (&&)
 command list 7-13
 description, use 7-14
 designated 7-40

Append
 ed procedure See Ed
 output append symbol See Output

Appending files 4-5

Appending See Output

Argument

 filename 7-2
 list creation 7-2
 mail commands 6-6
 number checking, \$# variable 7-9
 processing 7-12
 redirection argument location 7-6
 shell argument passing 7-12
 substitution sequence 7-13
 switch See Switch
 test command argument 7-25

Arithmetic

 See also BC

 expr command effect 7-26

askcc option See Mail

asksubject option See Mail

Asterisk (*)

 BC

 comment convention 8-10

 comment convention 8-9

 multiplication operator symbol 8-2

 multiplication operator symbol 8-3

 directory name, use avoidance 7-3

 filename, use avoidance 3-4

 filename wildcard 3-6

 mail

 character matching 6-6

 message saved, header notation 6-12

 message saved, header notation 6-13

 metacharacter 7-2

 metacharacter 7-40

 pattern matching functions 3-5

 pattern matching See metacharacter

 special shell variable 7-13

at command 4-16

At sign (@), mail 6-22

At sign (@), mail 6-30

atrm command 4-17

auto command, BC 8-13

autombox option See Mail

autoprint option See Mail

B

b command See Vi

-b option

 mail 6-23

Background job

 C-shell use See C-shell

Background process 4-17

Background process 4-18

Background process (*continued*)

- \$! variable 7-10
- ampersand (&) operator 4-17
- ampersand (&) operator 7-14
- ampersand (&) operator 7-40
- CNTRL-D immunity 7-14
- dial-up line
 - CNTRL-D effect 7-14
 - nohup command 7-14
- INTERRUPT immunity 7-14
- logout immunity 7-14
- QUIT immunity 7-14
- use restraint 7-14

Backslash (\)

BC

- comment convention 8-10
- comment convention 8-9
- line continuation notation 8-4

C-shell use See C-shell

ed See Ed

- escape character 2-3
- erasing 2-3
- line continuation notation 7-33
- metacharacter escape 7-3
- quoting 7-40

BACKSPACE key

- erasure function 2-3
- literal 2-3
- mail 6-5
- mail 6-8

Batch processing See Command

bc command

BC

- file reading, execution 8-9
- invocation 8-1
- calculation 4-22

BC

- addition operator
 - evaluation order 8-10
 - left to right binding 8-3
 - scale 8-12
 - scale 8-4
 - symbol (+) 8-3
- additive operators
 - See also Specific Operator
 - left to right binding 8-12
- alphabetic register See storage register
- arctan function
 - availability 8-1
 - loading procedure 8-9
- array
 - auto array 8-13
 - characteristics 8-10
 - identifier 8-10
 - identifier 8-14
 - name 8-6
 - named expression 8-11
 - one-dimensional 8-6
- assignment operator

BC (*continued*)

assignment operator (*continued*)

- designated, use 8-12
- evaluation order 8-10
- positioning effect 8-3
- symbol (=) 8-3

assignment statement 8-8

asterisk (*)

- comment convention 8-10
- comment convention 8-9
- multiplication operator symbol 8-2
- multiplication operator symbol 8-3

auto command 8-13

auto, keyword 8-10

auto statement

- built-in statement 8-14

backslash (\)

- comment convention 8-10
- comment convention 8-9
- line continuation notation 8-4

bases 8-4

bc command

- file reading, execution 8-9
- invocation 8-1

bc -l command 8-9

Bessel function

- availability 8-1
- loading procedure 8-9

BKSP key 8-1

braces ({})

- compound statement enclosure 8-14
- function body enclosure 8-5

brackets ([])

- array identifier 8-10
- auto array 8-13
- subscripted variable 8-6

break, keyword 8-10

break statement

- built-in statement 8-14

built-in statement 8-14

caret (^), exponentiation operator symbol 8-3

bol 8-3

command See bc command

comment convention 8-10

comment convention 8-9

compound statement 8-14

constant

- composition 8-10
- defined 8-11

construction

- diagram 8-9
- space significance 8-9

control statements 8-7

cos function

- availability 8-1
- loading procedure 8-9

define, keyword 8-10

define statement

- built-in statement 8-14

BC (continued)

- define statement (*continued*)
 - description, use 8-14
- demonstration run 8-1
- description 8-1
- division operator
 - left to right binding 8-12
 - left to right binding 8-3
 - scale 8-12
 - scale 8-5
 - symbol (/) 8-3
- equal sign (=)
 - assignment operator symbol 8-3
 - relational operator 8-13
 - relational operator 8-7
- equivalent constructions diagram 8-9
- evaluation sequence 8-2
- exclamation point (!)
 - relational operator 8-13
 - relational operator 8-7
- exit 8-1
- exit 8-2
- exponential function
 - availability 8-1
 - loading procedure 8-9
- exponentiation operator
 - right to left binding 8-12
 - right to left binding 8-3
 - scale 8-12
 - scale 8-5
 - symbol (^) 8-3
- expression
 - enclosure 8-11
 - evaluation order 8-10
 - named expression 8-11
 - statement 8-14
- for, keyword 8-10
- for statement
 - break statement effect 8-14
 - built-in statement 8-14
 - description, use 8-7
 - format 8-14
 - range execution 8-7
 - relational operator 8-13
- function call
 - defined 8-11
 - description 8-11
 - evaluation order 8-10
 - procedure 8-6
 - syntax 8-11
- function
 - argument absence 8-6
 - array 8-6
 - calling See function call
 - definition procedure 8-5
 - form 8-5
 - identifier 8-10
 - name 8-5
 - parameters 8-6

BC (continued)

- function (*continued*)
 - return statement See return statement
 - termination, return statement 8-15
 - variable automatic 8-5
- global storage class 8-13
-), relational operator 8-13
-), relational operator 8-7
- hexadecimal digit
 - ibase 8-4
 - obase 8-4
 - value 8-10
- ibase
 - decimal input setting 8-4
 - defined 8-11
 - initial setting 8-4
 - keyword 8-10
 - named expression 8-11
 - setting 8-4
 - variable 8-5
- identifier
 - array See array
 - auto statement effect 8-14
 - description 8-10
 - global 8-13
 - local 8-13
 - named expression 8-11
 - value 8-13
- if, keyword 8-10
- if statement
 - built-in statement 8-14
 - description, use 8-7
 - format 8-14
 - range execution 8-7
 - relational operator 8-13
- INTERRUPT key 8-1
- introduction 8-1
- invocation 8-1
- keywords designated 8-10
- l command, BC 8-9
- language features 8-8
- length
 - built-in function 8-11
 - keyword 8-10
- less-than sign (<), relational operator 8-13
- less-than sign (<), relational operator 8-7
- line continuation notation 8-4
- local storage class 8-13
- log function
 - availability 8-1
 - loading procedure 8-9
- math function library See bc -l command
- minus sign (-)
 - subtraction operator symbol 8-3
 - unary operator symbol 8-12

Index

BC (*continued*)
 minus sign (-) (*continued*)
 unary operator symbol 8-3
 modulo operator
 left to right binding 8-12
 left to right binding 8-3
 scale 8-12
 scale 8-5
 symbol (%) 8-3
 multiplication operator
 evaluation order 8-10
 left to right binding 8-12
 left to right binding 8-3
 scale 8-12
 scale 8-5
 symbol (*) 8-2
 symbol (*) 8-3
 multiplicative operators
 See also Specific Operator
 left to right binding 8-12
 named expression 8-11
 negative number, unary minus sign (-)
 8-3
 obase
 conversion speed 8-4
 defined 8-11
 description 8-4
 hexadecimal notation 8-4
 initial setting 8-4
 keyword 8-10
 named expression 8-11
 variable 8-5
 operator
 See also Specific Operator
 designated, use 8-3
 parentheses (())
 expression enclosure 8-11
 function identifier argument enclosure 8-10
 percentage sign (%), modulo operator
 symbol 8-3
 plus sign (+)
 addition operator symbol 8-3
 unary operator symbol 8-12
 program flow alteration 8-7
 quit command 8-1
 quit command 8-2
 quit, keyword 8-10
 quit statement
 BC exit 8-15
 built-in statement 8-14
 quoted string statement 8-14
 register See storage register
 relational operator
 designated 8-13
 designated 8-7
 evaluation order 8-10
 RETURN key 8-1
 return, keyword 8-10
 return statement

Index

BC (*continued*)
 return statement (*continued*)
 built-in statement 8-14
 description 8-15
 form 8-5
 scale command 8-5
 scale
 addition operator 8-12
 addition operator 8-4
 arctan function 8-9
 Bessel function 8-9
 built-in function 8-11
 command See scale command
 cos function 8-9
 decimal digit value 8-5
 defined 8-11
 description 8-4
 division operator 8-12
 division operator 8-5
 exponential function 8-9
 exponentiation operator 8-12
 exponentiation operator 8-5
 initial setting 8-5
 keyword 8-10
 length function 8-11
 length maximum 8-4
 log function 8-9
 modulo operator 8-12
 modulo operator 8-5
 multiplication operator 8-12
 multiplication operator 8-5
 named expression 8-11
 sin function 8-9
 square root effect 8-11
 square root effect 8-5
 subtraction operator 8-12
 subtraction operator 8-4
 value printing procedure 8-5
 variable 8-5
 semicolon (;), statement separation
 8-13
 semicolon (;), statement separation
 8-2
 sin function
 availability 8-1
 loading procedure 8-9
 slash (/), division operator symbol
 8-3
 space significance 8-9
 square root
 built-in function 8-11
 keyword 8-10
 result as integer 8-3
 scale procedure 8-5
 sqrt keyword 8-10
 statement
 See also Specific Statement
 entry procedure 8-8
 execution sequence 8-13
 separation methods 8-13

BC (*continued*)
 statement (*continued*)
 types designated 8-13
 storage classes 8-13
 storage register 8-3
 subscripted variable
 array See array
 description 8-6
 subscript See subscript
 subscript
 fractions discarded 8-6
 truncation 8-10
 value limits 8-6
 subtraction operator
 left to right binding 8-3
 scale 8-12
 scale 8-4
 symbol (-) 8-3
 syntax 8-1
 token composition 8-10
 truncation use when 8-5
 unary operator
 designated 8-11
 evaluation order 8-10
 left to right binding 8-11
 symbol (-) 8-3
 variable
 automatic 8-13
 automatic 8-5
 name 8-5
 subscripted See subscripted variable

 while, keyword 8-10
 while statement
 break statement effect 8-14
 built-in statement 8-14
 description, use 8-7
 execution 8-15
 range execution 8-7
 relational operator 8-13

 ~bcc escape See Mail
 Bessel function See BC
 /bin directory
 command search 7-2
 contents 3-4
 contents 7-23
 name derivation 7-23
 /usr/bin duplicate determination 7-32

 Binary file See File
 Binary logical and operator 7-25
 Binary logical or operator 7-25
 BINUNIQ shell procedure 7-32
 BKSP key
 BC 8-1
 command-line buffer editing 3-6
 BKSP
 vi cursor movement 5-13
 Block special device 4-11
 Bourne shell

Bourne shell (*continued*)
 TERM variable 5-38
 terminal type 5-38
 Braces ({})
 BC
 compound statement enclosure 8-14
 function body enclosure 8-5
 command grouping 7-18
 pipeline, command list enclosure 7-14
 variable
 conditional substitution 7-27
 enclosure 7-8
 Braces command ({}) 7-29
 Brackets ([])
 BC
 array identifier 8-10
 auto array 8-13
 subscripted variable 8-6
 directory name, use avoidance 7-3
 ed metacharacter See Ed
 filename, use avoidance 3-4
 metacharacter 7-2
 metacharacter 7-40
 pattern matching See metacharacter

 pattern-matching functions 3-6
 test command, use in lieu of 7-24
 break command
 for command control 7-17
 loop control 7-17
 shell built-in command 7-29
 special shell command 7-21
 while command control 7-17
 BREAK key
 program stopping 2-3
 terminal nonsense character removal 2-1
 Buffer See Ed
 Buffers See Vi 5-18

C

 c command See Ed
 C language
 BC
 comment convention similarity 8-9
 syntax agreement 8-1
 shell language 7-1
 -c option, shell invocation 7-28
 mail 6-23
 C shell
 TERM variable 5-38
 terminal type setting 5-38
 cal command 4-21
 Calculation
 See also BC
 example 4-22

Index

Calculator functions See BC
 calendar command 4-22
 Calendar reminder service 6-23
 Caret (^)
 BC, exponentiation operator symbol 8-3
 ed use See Ed
 mail, first message specification 6-11
 mail, first message specification 6-25
 mail, first message specification 6-5
 case command
 description, use 7-15
 exit status 7-16
 redirection 7-19
 shell built-in command 7-29
 Case delimiter symbol (;) 7-40
 Case significance 2-2
 Case-part 7-38
 cat command
 ed See Ed
 file
 combining 4-5
 contents display 2-2
 Cat
 command 4-5
 ~cc escape See Mail
 cd arg command 7-21
 cd command 4-11
 directory change 3-4
 directory change 7-10
 mail 6-16
 mail 6-25
 parentheses use 7-10
 time consumption minimization 7-30
 Changing password 4-2
 Changing terminal types 4-2
 Character class See Ed
 Character counting 4-16
 Character special device 4-11
 chmod command 4-13
 chmod command 4-14
 directory permission change 3-2
 file permission change 3-1
 chron option See Mail
 CNTRL-C, program stopping 2-3
 CNTRL-D
 background process immunity 7-14
 BC exit 8-1
 BC exit 8-2
 end-of-file 4-1
 logging out 2-4
 mail 4-20
 message sending 6-2
 message sending 6-7
 reply message termination 6-14
 reply message termination 6-9
 shell exit 6-16
 shell exit 7-18
 vi scroll 5-15

Index

CNTRL-F
 vi scroll 5-15
 CNTRL-G
 vi See Vi 5-8
 CNTRL-H, mail 6-5
 CNTRL-Q, output resumption 4-3
 CNTRL-S, output stopping 4-3
 CNTRL-U
 command-line buffer editing 3-6
 kill character 2-3
 line kill 4-3
 literal 2-3
 mail, line killing 6-5
 mail, line killing 6-8
 vi scroll 5-15
 Co command See Vi
 Colon command (:)
 description 7-21
 shell built-in command 7-29
 special shell command 7-21
 Colon (:)
 command See Colon command (:)
 mail
 command escape 6-19
 network mail 6-10
 PATH variable use 7-9
 variable conditional substitution 7-28
 vi use See Vi
 Command line
 ampersand (&) effect 3-7
 buffer defined 3-6
 defined 3-6
 entry 4-3
 erasure 4-3
 execution 7-13
 interpretation 3-6
 multiple commands entry 3-7
 options
 See also Specific Option
 designated 7-28
 pipeline, use in 7-14
 rescan 7-13
 RETURN key effect 4-3
 scanning sequence 7-13
 substitution 7-6
 case command, execution 7-16
 defined 7-13
 for command, execution 7-17
 grammar 7-38
 Command
 See also Specific Command
 background submittal 3-7
 batch processing See background submittal
 dash (-) use 3-4
 defined 7-13
 delimiter See Ed
 directory See /bin directory
 directory See Directory

Command (*continued*)

- ed commands See Ed
- enclosure in parentheses (()), effect 7-29
- environment 7-11
- execution 3-6
- execution 7-1
 - RETURN key required 2-1
 - sequence 4-17
 - time 7-29
- exit status See Exit status
- grammar 7-38
- grouping
 - exit status 7-19
 - parentheses (()) use 7-40
 - procedure 7-18
 - WRITEMAIL shell procedure 7-37
- keyword parameter 7-11
- line See Command line
- list See Command list
- lowercase letters 3-7
- mail commands summary 6-25
- multiple commands entry 3-7
- multiple commands entry 7-6
- name error 2-2
- output substitution symbol 7-40
- private command name 7-2
- program invocation 3-6
- public command name 7-2
- RETURN key required 2-1
- search
 - PATH variable 7-9
 - process 7-31
- separation symbol (;) 7-40
- shell, built-in commands designated 7-29
- simple command
 - defined 7-1
 - defined 7-13
 - grammar 7-38
- slash (/) beginning, effect 7-2
- special shell commands See Shell
- special shell commands See Specific Special Command
- substitution
 - back quotation marks (") 7-3
 - double quotation marks (") 7-3
 - procedure 7-6
 - redirection argument 7-4
- syntax 3-7
- typing error correction 2-3
- vi commands See Vi

Commands

- at 4-16
- atrm 4-17
- cal 4-21
- cat 4-5
- cd 4-11

Commands (*continued*)

- copy 4-10
- cp 4-6
- date 4-21
- diff 4-14
- diff3 4-14
- echo 4-14
- find 4-7
- head 4-5
- kill 4-19
- lc 4-8
- ln 4-7
- lpr 4-19
- mkdir 4-9
- more 4-4
- mv 4-6
- passwd 4-2
- ps 4-18
- pwd 4-10
- rm 4-6
- rmdir 4-9
- sort 4-15
- stty 4-3
- tail 4-5
- wc 4-16
- Communication See Mail
- Comparing files 4-14
- compose escapes 6-1
- Compose escapes See Mail
- Concatenate See cat command
- continue command
 - for command control 7-17
 - shell built-in command 7-29
 - special shell command 7-21
 - until command control 7-17
 - while command control 7-17
- Control characters
 - filename use restrictions 3-3
- Control command
 - See also Specific Control Command
 - redirection 7-19
- Copy command 4-10
- Copying a directory 4-10
- Copying files 4-6
- Copying See cp command
- COPYPAIRS shell procedure 7-32
- COPYTO shell procedure 7-33
- Counting, wc command 4-16
- cp command 4-6
- Creating a directory 4-9
- Creating a file 4-4
- csh command
 - C-shell invocation 10-1
- C-shell
 - & symbol
 - redirection use 10-6
 - alias command
 - multiple command use 10-6
 - number limitation 10-6

C-shell (*continued*)

- alias command (*continued*)
 - pipeline use 10-6
 - quoting 10-6
 - use 10-5
 - use 10-7
- command See alias command
- listing 10-7
- removal 10-8
- ampersand (&)
 - background job symbol 10-7
 - background job use 10-16
 - boolean AND operation implementation (&&) 10-10
 - if statement, avoidance 10-12
 - redirection symbol 10-6
- appending
 - noclobber variable effect 10-6
 - symbol (>>) 10-6
- argument
 - expansion 10-15
 - group specification 10-16
- argv variable
 - filename expansion prevention 10-11
 - script arguments contents 10-8
- arithmetic operations 10-10
- asterisk (*)
 - character matching 10-16
 - script notation 10-10
- background job
 - procedure 10-7
 - symbol (&) 10-7
 - termination procedure 10-7
- backslash (\)
 - if statement use 10-12
 - metacharacter cancellation 10-16
 - metacharacter escape 10-6
 - root parts separation from extension 10-16
- boolean AND operation implementation 10-10
- boolean OR operation implementation 10-10
- braces ({})
 - argument expansion use 10-15
 - argument grouping 10-16
- brackets ([])
 - character matching 10-16
- break command
 - foreach statement exit 10-13
 - loop break 10-11
 - while statement exit 10-13
- breaksw command
 - switch exit 10-13
- c command
 - reuse 10-3
- caret (^)

C-shell (*continued*)

- caret (^) (*continued*)
 - history substitution use 10-17
- character matching 10-16
- colon (:)
 - script modifier 10-12
 - substitution modifier use 10-17
- command prompt symbol (%) 10-1
- command substitution
 - string modification 10-12
- command
 - default argument supply 10-5
 - execution status 10-11
 - expansion 10-16
 - file See script
 - history list 10-3
 - input supply 10-13
 - location determination 10-7
 - location recomputation 10-2
 - multiple commands See commands, multiple
 - quoting 10-15
 - quoting, replacement 10-16
 - read only option 10-14
 - reading from file 10-8
 - repetition 10-7
 - repetition mechanisms 10-5
 - separation 10-16
 - separation symbol (;) 10-6
 - similarity, foreach command use 10-15
 - simplification 10-5
 - substitution symbol 10-17
 - termination testing 10-11
 - timing 10-8
 - transformation 10-5
- commands, multiple
 - alias use 10-6
 - single job 10-6
- comment metacharacter 10-17
- comment
 - script use 10-9
- continue command
 - loop use 10-11
- .cshrc file use 10-1
 - alias placement 10-5
- diagnostic output
 - direction 10-6
 - redirection See redirection
- directory
 - examination 10-2
 - listing 10-2
- disk usage 10-7
- dollar sign (\$)
 - last argument symbol 10-4
 - process number expansion 10-10
 - variable substitution symbol 10-9
 - variable substitution use 10-17

C-shell (*continued*)

- du command 10-7
- :e modifier 10-12
- echo option 10-14
- else-if statement use 10-12
- environment
 - printing 10-7
 - setting 10-7
- equal sign (=)
 - string comparison use (==), (=~) 10-10
- exclamation point (!)
 - history list substitution use 10-7
 - history mechanism invocation character use 10-4
 - history substitution use 10-17
 - string comparison use (!=), (!~) 10-10
 - syntax use 10-3
- execute primitive 10-11
- existence primitive 10-11
- expansion metacharacters designated 10-17
- expansion
 - control 10-14
- expression primitives 10-11
 - enclosure 10-16
 - evaluation 10-10
- extension extraction 10-12
- file overwriting
 - prevention 10-3
 - procedure 10-3
- file
 - appending 10-6
 - command content See script
 - enquiries 10-10
- filename metacharacters designated 10-16
- filename
 - expansion 10-15
 - expansion prevention 10-11
 - home directory indication 10-16
 - root extraction 10-12
 - scratch filename metacharacter 10-17
- foreach command 10-14
 - exit 10-13
 - script use 10-11
- goto label
 - script cleanup 10-14
- goto statement 10-13
 -)> redirection symbol 10-6
 -)> redirection symbol 10-16
- history command 10-5
 - use 10-7
- history list 10-3
 - command substitution 10-7

C-shell (*continued*)

- history list (*continued*)
 - contents display 10-7
- history mechanism invocation character 10-4
- history mechanism
 - alias, use 10-6
 - use 10-5
- history variable 10-1
- history
 - substitution symbol 10-17
- home variable 10-2
- if statement use 10-12
- ignoreeof variable 10-1
- ignoreeof variable 10-2
- input metacharacters designated 10-16
- input
 - execution procedure 10-9
 - variable substitution See variable substitution
- INTERRUPT key
 - background job, effect 10-7
- invocation procedure 10-1
- kill command
 - background job termination 10-7
- less-than sign (<)
 - redirection symbol 10-16
 - script inline data supply (<<) 10-14
 - variable substitution 10-10
- logging out
 - logout command use 10-1
 - procedure 10-1
 - shield 10-1
- .login file use 10-1
- logout command
 - use 10-1
 - use 10-7
- .logout file use 10-1
- loop
 - break 10-11
 - input prompt 10-15
 - variable use 10-15
- mail program
 - invocation 10-1
- mail variable 10-3
 - new mail notification 10-1
- metacharacter
 - cancellation 10-16
 - expansion metacharacter 10-17
 - filename metacharacter 10-16
 - input metacharacter 10-16
 - output metacharacter 10-16
 - quotation metacharacter 10-16
 - substitution metacharacter 10-17
 - syntactic metacharacter 10-16

C-shell (*continued*)

- metaxyntax
 - exclamation point (!) use 10-3
- minus sign (-)
 - option prefix 10-17
- modifiers 10-12
- n key
 - script error absence 10-10
 - script notation 10-10
- n option 10-14
- new program access 10-2
- noclobber variable 10-3
 - appending procedure 10-6
 - redirection symbols 10-6
- noglob variable
 - filename expansion prevention 10-11
- number sign (#)
 - C-shell comment symbol 10-14
 - C-shell comment symbol 10-9
 - C-shell comment use 10-17
 - scratch filename use 10-17
- onintr label
 - script cleanup 10-14
- option
 - metacharacter 10-17
- output metacharacters designated 10-16
- output
 - diagnostic output See diagnostic output
 - redirection See redirection
- parentheses (())
 - expression enclosure 10-16
- path variable 10-2
- pathname
 - component separation 10-16
- percentage sign (%)
 - command prompt symbol 10-1
- pipe symbol (|)
 - boolean OR operation implementation (|) 10-10
 - command separation 10-16
 - if statement, avoidance 10-12
 - redirection symbol 10-6
- pipeline
 - alias, use 10-6
- primitives See expression primitives
- printenv
 - environment printing 10-7
- process number expansion notation 10-10
- process number
 - listing 10-7
- prompt variable 10-7

C-shell (*continued*)

- ps command
 - process number listing 10-7
- question mark (?)
 - character matching 10-16
 - loop input prompt 10-15
- QUIT signal
 - background job, effect 10-7
- quotation mark, back (')
 - command substitution use 10-17
- quotation mark, double (") 10-14
 - metacharacters cancellation 10-16
- quotation mark, single (')
 - quoted string, effect 10-14
 - metacharacters cancellation 10-16
- quotation marks, back (")
 - command quoting 10-15
- quotation marks, double (")
 - string quoting 10-15
- quotation marks, single (')
 - alias quoting 10-6
 - script inline data quoting 10-14
- quotation metacharacters designated 10-16
- :r modifier 10-12
- read primitive 10-11
- redirection
 - diagnostic output 10-6
 - output 10-6
 - symbols designated 10-16
- rehash command 10-2
 - command location recomputation 10-7
- repeat command
 - command repetition 10-7
- root part
 - extension, separation 10-16
- script
 - clean up 10-14
 - colon (:) modifier 10-12
 - command input 10-13
 - comment required 10-14
 - description 10-8
 - example 10-11
 - execution 10-8
 - exit 10-14
 - inline data supply 10-13
 - interpretation 10-8
 - interruption catching 10-14
 - metanotation for inline data 10-13
 - modifiers 10-12
 - notations 10-10
 - range 10-10
 - variable substitution See variable substitution

C-shell (*continued*)
 semicolon (;)
 command separation 10-16
 command separation 10-6
 if statement, avoidance 10-12
 set command
 variable listing 10-2
 variable value assignment 10-2
 setenv command
 environment setting 10-7
 slash (/)
 pathname component separation 10-16
 source command
 command reading 10-8
 status variable 10-11
 string
 comparison 10-10
 quoting 10-15
 substitution metacharacters designated 10-17
 switch statement
 exit 10-13
 syntactic metacharacters designated 10-16
 then statement use 10-12
 tilde (~)
 home directory indication 10-16
 string comparison (=~), (!~) 10-10
 time command
 command timing 10-8
 time variable 10-1
 unalias command
 alias removal 10-8
 unset command
 variable removal 10-8
 unsetting procedure 10-3
 -v command line option 10-14
 variable substitution
 procedure 10-9
 variable
 component access 10-9
 component access notations 10-9
 definition removal 10-8
 environment variable setting 10-7
 expansion 10-15
 expansion 10-9
 listing 10-2
 loop use 10-15
 See also Specific Variable
 setting procedure 10-2
 substitution metacharacter 10-17
 substitution See variable substitution

C-shell (*continued*)
 variable (*continued*)
 use 10-2
 value assignment 10-2
 value assignment check 10-9
 verbose option 10-14
 while statement
 exit 10-13
 write primitive 10-11
 -x command line option 10-14 .cshrc file"
 C-shell use 10-1
 Current directory
 change 3-4
 procedure 4-11
 description 4-11
 printing 4-8
 shorthand name 3-5
 user residence 3-4
 Current line
 See Vi
 Cursor movement
 vi See Vi
 Cutting and pasting procedure See Ed

D

D command See Vi
 d command
 ed use See Ed
 mail, message deletion See Mail
 d0 command See Vi
 Dash (-)
 command option use 3-4
 filename, use avoidance 3-4
 switch use 3-7
 Dash (-), permission
 denial notation 4-12
 ordinary file notation 4-11
 date command 2-1
 Date command 4-21
 dd command See Vi
 ~dead escape See Mail
 Delete buffer See Vi
 DELETE key
 program stopping 2-3
 Deleting a file 4-6
 Deletion See d command
 vi procedure See Vi
 Delimiter See Ed
 Demonstration 2-1
 /dev directory
 contents 3-4
 /dev/console directory
 contents 3-4
 Device special file See Special file
 Device
 filename 3-3

Device (*continued*)
 filenamerequired 3-3
 pathname 3-3
 /dev/tty directory
 contents 3-4
 Diagnostic output See Output
 Dial-up line See Background process
 Diff command 4-14
 diff3 4-14
 Digit grammar 7-38
 Directory
 access permission See Permission

 /bin See /bin directory
 changing 4-11
 command See cd command
 composition 3-1
 copying 4-10
 creating 4-9
 C-shell use See C-shell
 listing 10-2
 current directory See Current directory

 description 3-1
 /dev See /dev directory
 diagram 3-2
 file See File
 filename
 required 3-3
 unique to directory 3-3
 /lib See /lib directory
 listing 4-9
 columns 4-8
 logging in result 3-2
 long listing 4-9
 name, metacharacter avoidance 7-3
 nesting 3-2
 parent directory See Parent directory

 pathname required 3-3
 permission notation 4-11
 permission See Permission
 protection 3-2
 recursive listing 4-9
 removing 4-9
 renaming 4-10
 search permission See Permission

 search
 optimum order 7-31
 PATH variable 7-31
 sequence change 7-2
 size effect 7-31
 time consumption 7-30
 size consideration 7-31
 /tmp directory 4-18
 /tmp See /tmp directory
 /tty See /tty directory
 user control 3-2
 /usr See /usr directory

Directory (*continued*)
 working directory See Current directory
 Displaying a file 4-4
 DISTINCT1 shell procedure 7-33
 Division See BC
 Division See Calculation
 Dollar sign (\$)
 ed use See Ed
 mail, final message specification 6-11
 mail, final message specification 6-25
 mail, final message specification 6-5
 positional parameter prefix 7-7
 positional parameter prefix 7-8
 PS1 variable default value 7-9
 variable prefix 7-8
 vi See Vi
 Dot command (.)
 description, use 7-19
 shell built-in command 7-29
 shell procedure alternate 7-23
 special shell command 7-21
 Dot (.)
 command See Dot command (.)
 ed use See Ed
 mail, current message specification 6-11
 mail, current message specification 6-5
 vi use See Vi
 Dot option See Mail
 Double quotation marks See Quotation marks, double (
 dp command See Mail
 DRAFT shell procedure 7-34
 dw command See Vi
 dw command See Vi 5-20

E

e command
 ed use See Ed
 mail 6-25
 mail 6-5
 mailR 6-15
 -e option, shell procedure 7-23
 echo command 4-14
 description, use 2-2
 description, use 7-25
 mail 6-25
 -n option effect 7-25
 syntax 7-25
 Ed
 a command
 append A-3
 append A-41
 backslash (\) characteristics A-27

Ed (*continued*)

- a command (*continued*)
 - dot (.) setting A-35
 - dot (.) setting A-41
 - global combination A-21
 - input termination A-25
 - input termination A-3
- abortion, q command A-41
- address arithmetic A-8
- ampersand (&)
 - literal A-33
- ampersand (&)
 - metacharacter A-32
 - substitution A-32
- append See a command
- asterisk (*), metacharacter A-23
- asterisk (*), metacharacter A-29
- at sign (@), script A-40
- backslash (\)
 - a command A-27
 - c command A-27
 - g command A-20
 - i command A-27
 - line folding A-21
 - literal A-26
 - metacharacter A-23
 - metacharacter A-25
 - metacharacter escape A-26
 - metacharacter escape A-33
 - multiline construction A-21
 - number string A-21
 - v command A-20
- backspace printing A-21
- brackets ([])
 - character class A-31
 - metacharacter A-23
 - metacharacter A-31
- buffer
 - description A-3
 - writing to file See w command
- c command
 - backslash (\) characteristics A-27
 - dot (.) setting A-17
 - dot (.) setting A-35
 - dot (.) setting A-41
 - global combination A-21
 - input termination A-17
 - line change A-16
 - line change A-41
- caret (^)
 - character class A-31
 - line beginning notation A-28
 - metacharacter A-23
 - metacharacter A-28
- cat command A-5
- change command See c command
- character class A-31
- character

Ed (*continued*)

- character (*continued*)
 - deletion at line beginning A-31
- command
 - See also Specific Command
 - combinations A-20
 - delimiter character A-27
 - description A-3
 - editing command See e command
 - form A-41
 - INTERRUPT key effect A-38
 - listing A-41
 - multicommand line restrictions A-12
 - summary A-41
- context search See search
- current line See dot (.)
- cutting and pasting
 - move command See m command
- procedures A-38
- d command
 - deletion A-10
 - deletion A-41
 - dot (.) setting A-35
 - dot (.) setting A-41
- deletion See d command
- delimiter
 - character choice A-27
- description A-1
- dollar sign (\$)
 - last line notation A-11
 - last line notation A-28
 - last line notation A-7
 - line end notation A-27
 - line end notation A-28
 - metacharacter A-23
 - metacharacter A-27
 - multiple functions A-28
- dot (.)
 - current line notation A-8
 - description A-9
 - determination A-35
 - search setting A-14
 - search setting A-42
 - substitution setting A-12
 - symbol (.) A-25
 - symbol (.) A-9
 - value determination A-10
 - value determination A-42
- duplication See t command
- e command A-41
- e command A-5
- edit See e command
- entry A-2
- equals sign (=)
 - dot value printing (.=) A-10

Ed (*continued*)
 equals sign (=) (*continued*)
 dot value printing (.=) A-42
 last line value printing A-42
 escape command (!) A-23
 escape command (!) A-42
 exclamation point (!), escape command A-23
 exit See q command
 f command A-41
 f command A-6
 file
 insertion into another file A-38
 writing out A-39
 filename
 change A-6
 recovery A-6
 remembered filename printing A-41
 remembered filename printing A-6
 folding A-21
 g command
 a command combination A-21
 backslash (\) use A-20
 c command combination A-21
 command combinations A-19
 command combinations A-20
 dot (.) setting A-20
 i command combination A-21
 line number specifications A-20
 multiline construction A-21
 s command combination A-19
 s command combination A-41
 search, command execution A-19
 search, command execution A-41
 substitution A-13
 substitution A-23
 trailing g A-23
 global command See g command
 global command See v command
), tab notation A-21
 grep command A-23
 hyphen (-), character class A-31
 i command
 backslash (\) characteristics A-27
 dot (.) setting A-17
 dot (.) setting A-35
 dot (.) setting A-41
 global combination A-21
 input termination A-25
 insertion A-16
 insertion A-41

Ed (*continued*)
 in-line input scripts 7-34
 input
 termination A-17
 termination A-25
 termination A-3
 insert command See i command
 INTERRUPT key
 command execution effect A-38
 dot (.) setting A-38
 print stopping A-7
 introduction A-1
 invocation A-2
 j command, line joining A-34
 k command, line marking A-22
 l command
 folding A-21
 line listing A-21
 line listing A-41
 nondisplay character printing A-21
 number string A-21
 s command combination A-24
 less-than sign (<)
 backspace notation A-21
 line beginning
 character deletion A-31
 notation A-28
 line end
 notation A-27
 line number
 0 as line number A-38
 combinations A-8
 summary A-41
 line
 beginning See line beginning
 break See splitting
 folding A-21
 joining A-34
 marking A-22
 moving See m command
 number See line number
 rearrangement A-34
 splitting A-33
 writing out A-39
 list See l command
 m command
 dot (.) setting A-18
 dot (.) setting A-41
 line moving A-18
 line moving A-41
 warning A-19
 mail system See Mail
 marking See k command
 metacharacter
 ampersand (&) A-32
 asterisk (*) A-23
 asterisk (*) A-29
 backslash (\) A-23

Ed (continued)

metacharacter (continued)

backslash (\) A-25
 brackets ([]) A-23
 brackets ([]) A-31
 caret (^) A-23
 caret (^) A-28
 character class A-31
 combination A-29
 dollar sign (\$) A-23
 dollar sign (\$) A-27
 escape A-26
 escape A-33
 period (.) A-23
 period (.) A-24
 search A-31
 slash (/) A-23
 star (*) A-23
 star (*) A-29

minus sign (-), address arithmetic
 A-8

move

command See m command
 line marking A-22

multicommand line restrictions A-12

new line

substitution A-33

nondisplay character printing A-21

p command

dot (.) setting A-38
 multicommand line A-12
 printing A-41
 printing A-7
 s command combination A-24

pattern search See search

period (.)

a command input termination A-25

a command input termination A-3

c command input termination A-17

character substitution A-24

dot symbol See Dot (.)

i command input termination A-25

literal A-26

metacharacter A-23

metacharacter A-24

s command, effect A-24

script problems A-40

search problems A-23

troff command prefix A-19

plus sign (+), address arithmetic A-8

print

command See p command

line folding A-21

RETURN key effect A-10

stopping A-7

q command

abortion use A-41

Ed (continued)

q command (continued)

quit session A-4

quit session A-41

w command combination A-41

question mark (?)

exit warning A-3

search error message (?) A-14

search repetition (??) A-16

search, reverse direction (? ?) A-14

search, reverse direction (? ?) A-42

write warning A-5

quit See q command

quotation marks, single (")

line marking A-22

r command

dot (.) setting A-36

dot (.) setting A-41

file insertion A-38

positioning without address A-39

read file A-41

read file A-6

reading See r command

regular expression

description A-23

metacharacter list A-23

RETURN key, printing A-41

s command

ampersand (&) A-32

character match A-24

description, use A-11

description, use A-41

dot (.) setting A-12

dot (.) setting A-35

dot (.) setting A-41

g command combination A-13

g command combination A-19

g command combination A-41

l command combination A-24

line number A-24

new line A-33

p command combination A-24

search combination A-14

text removal A-12

trailing g A-23

undoing A-21

v command combination A-19

script A-40

search

dot (.) setting A-42

error message (?) A-14

Ed (*continued*)

- search (*continued*)
 - forward search (/ /) A-13
 - forward search (/ /) A-42
 - global search See g command
 - global search See v command
 - metacharacter problems A-23
 - next occurrence description A-14
 - procedure A-13
 - repetition (/ /), (??) A-16
 - reverse direction (? ?) A-14
 - separator A-37
 - substitution combination A-14
- sed command A-23
- semicolon (;)
 - dot (.) setting A-37
 - search separator A-37
- shell
 - escape See escape command (!)
- slash (/)
 - delimiter A-27
 - literal A-26
 - metacharacter A-23
 - search forward (/ /) A-13
 - search forward (/ /) A-42
 - search repetition (/ /) A-16
- special character See metacharacter
- spelling correction See s command
- star (*), metacharacter A-23
- star (*), metacharacter A-29
- substitution
 - command See s command
- t command
 - dot (.) setting A-42
 - transfer line A-22
 - transfer line A-42
- tab printing A-21
- tbl command A-39
- termination See q command
- text
 - removal See s command
 - saving A-4
- transfer See t command
- troff command printing A-19
- typing error correction See s command
- u command
 - undo A-21
 - undo A-42
- undo See u command
- v command
 - a command combination A-21
 - backslash (\) use A-20
 - c command combination A-21
 - command combinations A-19

Ed (*continued*)

- v command (*continued*)
 - command combinations A-20
 - command combinations A-21
 - dot (.) setting A-20
 - global search, substitute A-19
 - global search, substitute A-42
 - i command combination A-21
 - line number specifications A-20
 - s command combination A-19
- w command
 - description, use A-4
 - dot (.) setting A-36
 - dot (.) setting A-42
 - e command combination A-41
 - file write out A-39
 - frequent use advantages A-36
 - line write out A-39
 - write out A-39
 - write out A-4
 - write out A-42
- write out
 - command See w command
 - warning A-5
- EDFIND shell procedure 7-34
- ~editor escape See Mail
- Editor See Ed
- EDITOR string, mail 6-21
- EDITOR string, mail 6-29
- EDLAST shell procedure 7-34
- egrep See grep command
- elif clause See if command
- else clause See if command
- Else-part grammar 7-38
- Empty grammar 7-38
- Equal sign (=)
- BC
 - assignment operator symbol 8-3
 - relational operator 8-13
 - relational operator 8-7
- ed use See Ed
- mail, message number printing 6-12
- mail, message number printing 6-25
- variable
 - conditional substitution 7-28
 - string value assignment 7-7
- errdirect file 7-20
- Error output redirection 7-27
- ESCAPE key
 - vi See Vi
- Escape string, mail 6-22
- Escape string, mail 6-29
- /etc/termcap file 4-3
- eval command
 - command line rescan 7-13
 - shell built-in command 7-29
- Ex, ed similarity A-1
- Exclamation point (!)
 - BC, relational operator 8-13

Exclamation point (!) (*continued*)
 BC, relational operator 8-7
 C-shell use See C-shell
 ed use See Ed
 mail
 network mail 6-10
 shell command execution 6-15
 shell command execution 6-18
 shell command execution 6-25
 unary negation operator 7-25
 vi See Vi
 exec arg command 7-22
 exec command 7-29
 execmail, mail 6-30
 Exit code See \$? variable
 exit command
 shell built-in command 7-29
 shell exit 7-18
 special shell command 7-21
 Exit status
 \$? variable 7-9
 case command 7-16
 cd arg command 7-21
 colon command (:) 7-21
 command grouping 7-19
 false command 7-26
 if command 7-15
 read command 7-22
 true command 7-26
 until command 7-16
 wait command 7-22
 while command 7-16
 Exponentiation See BC
 Exponentiation See Calculation
 export command
 shell built-in command 7-29
 variable
 example 7-9
 listing 7-11
 setting 7-11
 expr command 7-26

F

F command, mail 6-15
 F command, mail 6-26
 F command, mail 6-9
 f command
 ed use See Ed
 mail 6-14
 mail 6-26
 mail 6-8
 mail 6-9
 -f option, mail 6-23
 -f option, mail 6-7

false command 7-26
 fgrep See grep command
 fi command
 if command end 7-15
 mail 6-26
 File descriptor
 description, use 7-4
 redirection 7-27
 redirection 7-4
 File permission
 changing 4-13
 File permissions, listing 4-9
 File system
 defined 3-3
 diagram 3-3
 organization 3-3
 File
 access
 control 3-1
 last access time 3-1
 permission See Permission
 addition See creation
 alphabetizing See sort
 appending 4-5
 attributes 3-1
 binary file 3-1
 combining 4-5
 composition 3-1
 copying 4-6
 creating 4-4
 with vi 5-2
 creation
 MKFILES shell procedure 7-36
 permission See Permission
 time 3-1
 write permission control 3-2
 defined 3-1
 deleting 4-6
 deletion
 write permission control 3-2
 descriptor See File descriptor
 directory See Directory
 displaying 4-4
 displaying 4-5
 editing See Vi
 filename See Filename
 grammar 7-38
 inode number See Inode number
 linking 4-7
 listing 3-2
 mail system files See Mail
 manipulation 4-3
 modification time 3-1
 moving 4-5
 moving 4-6
 name See Filename
 paginating 4-20
 pathname, printing 4-10
 pathname required 3-3

File (*continued*)

pattern search See Ed
 pattern search See grep command

 pattern search See Pattern matching
 facility
 permission See Permission
 permissions 4-11
 pipe interchange 7-33
 printing See Lineprinter
 protection 3-1
 removal 4-6
 renaming 4-6
 comparing 4-14
 scratch file directory 3-4
 shell procedure creation 7-22
 size in bytes 3-1
 sorting 4-15
 special file See Special file
 temporary file See Temporary file

 textual contents determination 7-37
 types designated 3-1
 variable file creation See Variable

Filename

argument 7-2
 asterisk (*) wildcard 3-6
 characters use restrictions 3-3
 description 3-3
 ed See Ed
 example designated 3-4
 long listing 4-9
 question mark (?) representation 3-6
 required 3-1
 required 3-3
 unique to directory 3-3

Filter

description 7-5
 order consideration 7-30

find command 4-7

Finding a file 4-7

finger command 4-19

Flag See Option

for command

break command effect 7-17
 continue command effect 7-17
 description, use 7-16
 redirection 7-19

shell built-in command 7-29

for loop, argument processing 7-12

Foreground process 4-17

Foreground process 4-18

fork command 7-29

FSPLIT shell procedure 7-35

Full pathname See Pathname

G

g command See Ed
 vi See Vi
 Global
 ed use See Ed
 variable check 7-24
 goto command
 See G command 5-4
)
 BC, relational operator 8-13
 BC, relational operator 8-7
 file combination 4-5
 output redirection 3-8
 PS2 variable default value 7-9
 redirection symbol 2-2
 redirection symbol 7-40
 grep command 4-15
 ed See Ed
 Group permission See Permission

H

h command
 mail 6-12
 mail 6-26
 mail 6-6
 H command
 vi use See Vi
 H flag, mail 6-12
 head command 4-5
 headers command See Mail
 ~headers escape See Mail
 history command
 C-shell use 10-5
 ho command See Mail
 Home directory 4-11
 HOME variable
 conditional substitution 7-28
 description 7-8

I

i command See Ed
 -i option
 mail 6-22
 mail 6-23
 mail 6-30
 mail 6-7
 shell invocation 7-28
 if command
 COPYTO shell procedure 7-33
 description, use 7-14
 exit status 7-15
 fi command required 7-15

if command (*continued*)
 multiple testing procedure 7-15
 nesting 7-15
 redirection 7-19
 shell built-in command 7-29
 test command 7-24
 IFS variable 7-8
 ignore option See Mail
 ignorecase option See Vi 5-27
 In-line input document See Input
 Inode number
 defined 3-2
 link See Link
 ls command 3-2
 required for file 3-1
 required for file 3-2
 Input
 ed See Ed
 grammar 7-38
 in-line input
 document 7-26
 EDFIND shell procedure 7-34
 keyboard origin 3-8
 redirection See Redirection
 standard input file 7-4
 termination 4-1
 Insert mode See Vi
 Insertion See Ed
 Internal field separator
 shell scanning sequence 7-13
 specificaiton by IFS variable 7-8
 INTERRUPT key
 background process immunity 7-14
 BC 8-1
 command-line buffer cancellation 3-7
 ed use See Ed
 foreground process killing 4-18
 logging in, nonsense character removal 2-1
 askcc switch 6-20
 message abortion 6-21
 message abortion 6-8
 program stopping 2-3
 Interrupt
 handling methods 7-19
 key See INTERRUPT key
 Invocation flag See Option
 Item grammar 7-38

J

j command See Ed

vi use See Vi

j command See Ed (*continued*)

K

k command See Ed
 vi use See Vi
 -k option, shell procedure 7-24
 Keyword parameter
 description 7-11
 -k option effect 7-24
 Kill character See CNTRL-U
 kill command 4-18
 kill command 4-19
 C-shell use See C-shell
 Killing a process 4-18

L

l command 4-9
 ed use See Ed
 mail 6-14
 mail 6-26
 vi use See Vi
 -l option
 function 3-7
 lc command 4-8
 listing 2-2
 Less-than sign (<)
 BC, relational operator 8-13
 BC, relational operator 8-7
 redirection symbol 7-40
 Less-than symbol (<)
 input redirection 3-8
 /lib directory
 contents 3-4
 line command
 shell variable value assignment 7-6
 Line
 beginning See Ed
 counting See wc command
 writing out See Ed
 linenumbr option See Vi
 Line-oriented commands See Vi 5-9
 Lineprinter
 command See lpr command
 file printing 4-19
 queue information 4-19
 queue information 4-20
 Link
 command See ln command
 defined 3-2
 description 4-7
 long listing 4-9
 Linking files 4-7
 list command
 mail 6-26
 list option See Vi

LISTFIELDS shell procedure 7-35
 Listing directory contents 4-8
 Listing See l command
 Listing See lc command
 ln command 4-7
 Logging in 4-1
 nonsense character removal 2-1
 procedure 2-1
 prompt character 2-1
 terminal behavior remedy 2-3
 type-ahead restriction 2-3
 Logging out
 background process immunity 7-14
 procedure 2-4
 procedure 4-1
 shell termination 7-18
 terminal behavior remedy 2-3
 Login directory
 defined 7-8
 new user 2-1
 .login file
 C-shell use 10-1
 Login message 2-1
 Login name
 new user 2-1
 Login
 procedure 4-1
 logout command
 C-shell use 10-1
 .logout file
 C-shell use 10-1
 Looping
 break command 7-17
 continue command 7-17
 control 7-17
 expr command 7-26
 false command 7-26
 for command 7-16
 iteration counting procedure 7-26
 time consumption 7-29
 true command 7-26
 unconditional loop implementation 7-26
 until command 7-16
 while command 7-16
 while loop 7-32
 lpr command
 file printing 4-19
 mail
 -m option 6-23
 message printing 6-14
 message printing 6-26
 pipe 4-20
 pr command combination 4-20
 ls command
 echo * use in lieu of 7-25
 function 3-2

 inode number use 3-2

ls command (*continued*)

M

m command
 ed See Ed
 mail 6-14
 mail 6-26
 M flag See Mail
 -m option, mail 6-23
 magic option See Vi
 mail command See Mail
 MAIL variable 7-8
 Mail
 ? command See help command (?)

 ~: See command escape (~:)
 ~? See help escape (~?)
 ~ See shell escape (~)
 ~~ See tilde quote escape (~~)
 a command See alias
 accumulation 6-24
 Alias 6-25
 a command 6-15
 a command 6-25
 a command 6-9
 display 6-9
 network mail 6-10
 personal 6-20
 personal 6-9
 R command 6-10
 system-wide 6-20
 askcc option 6-10
 askcc option 6-20
 askcc option 6-29
 asksubject option 6-20
 asksubject option 6-29
 asterisk (*)
 character matching 6-6
 message saved, header notation 6-12
 message saved, header notation 6-13
 at sign (@), ignore switch echo 6-22
 at sign (@), ignore switch echo 6-30
 autombox option
 description, use 6-22
 description, use 6-29
 effect 6-13
 H flag 6-12
 ho command 6-14
 autoprint option 6-20
 autoprint option 6-29
 ~b escape 6-17
 -b option 6-23
 BACKSPACE key 6-5
 BACKSPACE key 6-8
 ~bcc escape 6-28

Mail (*continued*)

Bcc field See blind carbon copy field

blind carbon copy field

description 6-4

editing 6-17

editing 6-18

escape See ~bcc escape

box See Mailbox

~c escape 6-17

-c option 6-23

carbon copy field

additions prompt 6-10

blind See blind carbon copy field

description 6-4

display 6-3

editing 6-18

escape See ~c escape

escape See ~cc escape

option See askcc option

R command effect 6-9

caret (^), first message specification 6-11

caret (^), first message specification 6-25

caret (^), first message specification 6-5

~cc escape 6-28

cc field See carbon copy field

cd command 6-16

cd command 6-25

chron option 6-20

chron option 6-29

CNTRL-D

message reply 6-14

message reply 6-9

message sending 6-7

CNTRL-H, backspace 6-5

CNTRL-U, line killing 6-5

CNTRL-U, line killing 6-8

colon (:) escape See command escape (~:)

network mail 6-10

command escape (~:) 6-19

command escape (~:) 6-28

command line options 6-23

command mode

description, use 6-5

help command 6-10

options setting 6-10

command

See also Specific Command

descriptions 6-10

escape See command escape (~:)

invocation 6-10

mail command See mail command

summary 6-25

syntax 6-6

compose escape (~) 6-28

Mail (*continued*)

compose escapes

See also Specific Escape

compose mode exit 6-5

edit mode entry 6-5

heading escapes 6-17

listing 6-2

listing 6-9

m command 6-14

reply 6-14

summary 6-28

tilde (~) component 6-9

compose mode

compose escapes See compose escapes

description, use 6-5

edit mode entry 6-5

entry from command mode 6-8

entry from shell 6-8

tilde escapes See compose escapes

concepts 6-3

C-shell

new mail notification 10-1

d command 4-21

d command 6-12

d command 6-25

d command 6-3

d command 6-6

d command 6-8

~d escape 6-18

~dead escape 6-28

dead.letter file

escape See ~d escape

nosave switch effect 6-21

undelivered message receipt 6-8

deletion See message

distribution list creation 6-9

dollar sign (\$), final message

specification 6-11

dollar sign (\$), final message

specification 6-25

dollar sign (\$), final message

specification 6-5

dot (.), current message specification

6-11

dot (.), current message specification

6-5

dot option 6-21

dot option 6-29

dp command 6-13

dp command 6-25

e command 6-15

e command 6-25

~e escape 6-17

echo command 6-25

~editor escape 6-28

editor escape See ~e escape

editor escape See ~v escape

EDITOR string 6-21

Mail (*continued*)

- EDITOR string 6-29
- entry 6-7
- equal sign (=), message number printing 6-12
- equal sign (=), message number printing 6-25
- escape string 6-22
- escape string 6-29
- exclamation point (!)
 - network mail 6-10
 - shell command execution 6-15
 - shell command execution 6-18
 - shell command execution 6-25
- execmail option 6-21
- execmail option 6-30
- exit
 - q command 4-21
 - q command 6-13
 - q command 6-26
 - q command 6-3
 - q command 6-7
 - x command 6-13
 - x command 6-26
- f command 6-14
- F command 6-15
- F command 6-26
- f command 6-8
- F command 6-9
- f option 6-23
- f option 6-7
- fi command 6-26
- file switch See -f option
- files designated 6-24
- forwarding
 - messages not deleted 6-13
 - procedure See F command
- h command 6-12
- h command 6-26
- h command 6-6
- ~h escape 6-18
- H flag, message saving 6-12
- header
 - characteristics 6-12
 - command See h command
 - defined 6-6
 - display 6-2
 - display 6-6
 - display 6-7
 - listing 6-26
 - windows 6-12
 - windows 6-6
- ~headers escape 6-28
- heading
 - compose escapes 6-17
 - composition 6-4
- help command (?) 6-10

Mail (*continued*)

- help command (?) 6-3
- help escape (~?) 6-16
- help escape (~?) 6-28
- help escape (~?) 6-9
- ho command
 - description 6-14
 - H flag 6-12
 - message saving 6-26
- hold command See ho command
- i option 6-22
- i option 6-23
- i option 6-30
- i option 6-7
- ignore switch See -i option
- INTERRUPT key
 - message abortion 6-21
 - message abortion 6-8
 - recipient list 6-20
- introduction 6-1
- invocation, -i option 6-7
- l command 6-14
- l command 6-26
- line killing 6-5
- line killing 6-8
- list command 6-26
- lpr command
 - m option 6-23
 - message printing 6-14
 - message printing 6-26
- m command 6-14
- m command 6-26
- ~M escape 6-18
- M flag, message saving 6-12
- m option 6-23
- mail command
 - command mode entry 6-5
 - command mode entry 6-7
 - compose mode entry 6-8
 - help 6-3
 - message reading 6-2
 - message reading 6-8
 - message sending 6-2
 - message sending 6-26
- mail escapes See ~M escape
- mailbox See Mailbox
- .mailrc file
 - alias contents 6-15
 - distribution list creation 6-9
 - example 6-20
 - options setting 6-10
 - set command 6-15
 - unset command 6-15
- mb command 6-14
- mb command 6-26
- mbox command See mb command
- mchron option 6-30
- ~message escape 6-28
- ~Message escape 6-29

Mail (continued)

message number
 command 6-12
 command 6-25
 message printing 6-8
 printing 6-12
 printing 6-25
 types 6-5
 message
 abortion 6-21
 abortion 6-7
 abortion 6-8
 advancement 6-25
 advancement 6-8
 body 6-4
 composition 6-4
 deletion 4-21
 deletion 6-12
 deletion 6-25
 deletion 6-3
 deletion 6-6
 deletion 6-8
 deletion undoing 6-13
 description 6-4
 display 4-21
 editing 6-15
 editing 6-23
 editing 6-25
 editing 6-8
 file inclusion 6-18
 forwarding See forwarding
 header See header
 heading See heading
 insertion into new message 6-18
 list See message-list
 argument, multiple messages 6-9
 composition 6-5
 full message-list description 6-6
 listing 6-3
 number See message number
 printing See printing
 range description 6-5
 reading 6-2
 reading 6-8
 reading into file 6-7
 reply See reply
 saving See saving
 sending See sending
 size 6-16
 size 6-27
 specification 6-9
 undeletion 6-8
 metacharacters 6-11
 metacharacters 6-5
 metoo option 6-21
 metoo option 6-30
 minus sign (-), message advancement 6-25
 network mail 6-10

Mail (continued)

noisy phone line 6-7
 nosave option 6-21
 nosave option 6-30
 number command See message number

 options
 See also Specific Option
 command line options 6-23
 setting 6-10
 summary 6-29
 switch option setting 6-15
 organization 6-24
 p command
 message printing 6-11
 message printing 6-26
 message printing 6-3
 message printing 6-6
 syntax 6-6
 ~p escape 6-17
 page option 6-22
 period (.), dot use See dot (.)
 phone line noise 6-7
 plus sign (+), message advancement 6-25
 ~print escape 6-29
 printing
 command See lpr command
 command See p command
 escape See ~p escape
 lineprinter See lpr command
 procedure 6-6
 procedure 6-8
 top five lines See t command
 programs designated 6-24
 prompt 4-21
 prompt 6-2
 q command
 exit 4-21
 exit 6-13
 exit 6-26
 exit 6-3
 exit 6-7
 message abortion 6-21
 question mark (?)
 command summary printing 6-25
 compose escape help See help escape (~?)
 help command 6-10
 quiet option 6-21
 quiet option 6-30
 ~quit escape 6-29
 R command
 alias effect 6-10
 compose mode entry 6-8
 message reply 6-14
 message reply 6-27
 r command
 message reply 6-8

Mail (continued)

- r command (continued)
 - message reply 6-9
 - ~r escape 6-18
 - R option 6-23
 - ~read escape 6-29
 - read escape See ~d escape
 - read escape See ~r escape
 - reading 4-21
 - recipient list, name addition 6-17
 - record string 6-22
 - record string 6-30
 - reminder service 4-22
 - reminder service 6-23
 - Reply command See R command
- return receipt request field 6-4
- s command
 - flag 6-12
 - message saving 6-13
 - message saving 6-27
 - system mailbox, message deletion 6-13
- ~s escape 6-17
- s option 4-20
- s option 6-23
- saving
 - asterisk (*) notation 6-13
 - automatic 6-12
 - command See s command
 - flag 6-12
 - ho command 6-26
 - M flag 6-12
 - message display 6-3
 - s command 6-13
 - s command 6-27
 - system mailbox 6-7
 - w command 6-13
 - w command 6-27
- se command See set command
- sending 4-20
 - cancellation impossible 6-2
 - multiple recipients 6-7
 - network mail 6-10
 - procedure 6-7
 - to self 6-2
- session abortion 6-8
- set command
 - description, use 6-15
 - description, use 6-27
 - option control 6-29
- set options defined 6-20
- sh command 6-15
- sh command 6-27
- shell commands 6-15
- shell escape (␣) 6-18
- SHELL string 6-22
- SHELL string 6-30
- si command 6-16
- si command 6-27

Mail (continued)

- so command 6-16
- so command 6-27
- source command See so command
- special characters See metacharacters
- startup file 6-20
- string option
 - setting 6-15
 - summary 6-29
- ~subject escape 6-29
- subject escape See ~s escape
- subject field 6-3
- subject field 6-4
- subject switch See asksubject option
- subject switch See -s option
- switch See Option
- system composition 6-24
- system mailbox, message retention 6-7
- t command
 - message top printing 6-12
 - message top printing 6-27
 - message top printing 6-9
 - toplines option 6-12
- ~t escape 6-17
- tilde escapes See compose escapes
- tilde quote escape (~) 6-19
- tilde quote escape (~) 6-28
- ~to escape 6-29
- to field
 - mandatory 6-4
 - R command effect 6-9
- top command See t command
- toplines option 6-30
- toplines string 6-22
- u command 6-13
- u command 6-27
- u command 6-6
- u command 6-8
- u option 6-23
- undeletion See u command
- unset command
 - description, use 6-15
 - description, use 6-27
 - option control 6-29
- v command 6-15
- v command 6-27
- v command 6-5
- ~v escape 6-17
- variable See MAIL variable
- vertical bar (|) escape See shell escape (␣)
- ~visual escape 6-29
- VISUAL string 6-21
- VISUAL string 6-30
- w command

Mail (*continued*)w command (*continued*)

message write out 6-13
 message write out 6-27
 system mailbox, message deletion
 6-13

~w escape 6-18

~write escape 6-29

write escape See ~w escape

write out See w command

x command

exit 6-13

exit 6-26

session abortion 6-8

you have mail message 2-1

Mailbox

cleaning out 6-24

command 6-14

reading in 6-7

system mailbox 6-4

user mailbox

filename 6-4

message saving notation 6-12

Make directory See mkdir command

Marking See Ed

mb command See Mail

mbox command See Mail

mchron option

mail 6-30

mesg option See Vi

~message escape See Mail

Metacharacter

asterisk (*) 7-40

brackets ([]) 7-40

directory name use avoidance 7-3

escape 7-3

list designated 7-40

mail 6-11

mail 6-5

question mark (?) 7-40

redirection restriction 7-4

metoo option See Mail

Minus sign (-)

BC

subtraction operator symbol 8-3

unary operator symbol 8-12

unary operator symbol 8-3

mail, message advancement 6-25

redirection effect 7-26

subtraction operator symbol 8-3

variable conditional substitution 7-27

mkdir command 4-9

MKFILES shell procedure 7-36

more command 4-4

Move See mv command

Multiple way branch See case command

Multiplication See BC

mv command 4-5

mv command 4-6

directory moving 4-10

mv command (*continued*)

N

n command See Vi

-n option

echo command 7-25

shell procedure 7-24

Name grammar 7-38

Name special file 4-12

Named pipe 4-12

newgrp command

description 7-22

shell built-in command 7-29

special shell command 7-22

Newline substitution See Ed

next command See Vi 5-34

nohup command 7-14

nosave option See Mail

nu command See Vi 5-18

Null command See Colon command (:) 7-36

NULL shell procedure 7-36

Number sign (#), comment symbol
 7-40

O

-o operator 7-25

(o), write command message end
 4-21

(oo), write command message end
 4-21

Operator See BC

Option

See also Specific Option

configuration 3-7

DRAFT shell procedure 7-34

grouping 3-7

invocation flags 7-28

mail options See Mail

multiple options

grouping See grouping

separate listing 3-7

position 3-7

tracing, \$- variable 7-10

Options

terminal 4-3

vi options See Vi

Ordinary file See File

Or-if operator (|)

command list 7-13

description, use 7-14

designated 7-40

Output

> 7-4

> 7-40

appending

Output (*continued*)

- appending (*continued*)
 - procedure 3-8
 - symbol (>>) 3-8
- control 4-3
 -) 7-40
- diagnostic output file 7-4
- error redirection 7-27
- file reception 2-2
- grammar 7-38
- redirection 2-2
- redirection 4-5
- redirection See Redirection
- resumption 4-3
- standard error file See diagnostic out-
put file
- standard output file 7-4
- terminal screen destination 3-8

P

p command

- ed use See Ed
- mail
 - message printing 6-11
 - message printing 6-26
 - message printing 6-3
 - message printing 6-6
 - syntax 6-6

page option See Mail

Parent directory

- description 3-4
- shorthand name 3-5

Parentheses (())

BC

- expression enclosure 8-11
- function identifier argument enclo-
sure 8-10

- command grouping 7-18
- command grouping 7-29
- command grouping 7-40
- pipeline, command list enclosure 7-14
- test command operator 7-25

passwd command 4-2

Password

- changing 4-2
- invisible on screen 2-1,
- logging in 2-1
- new user 2-1

PATH variable

- conditional substitution 7-28
- C-shell use See C-shell
- description 7-9
- directory search
 - effect 7-31
 - sequence change 7-2

Pathname

- absolute pathname

Pathname (*continued*)absolute pathname (*continued*)

- example 3-4
- required 3-3
- slash (/) significance 3-4
- unique to system 3-3

defined 3-4

full pathname See absolute pathname

relative pathname

- defined 3-4
- example designated 3-4
- structure 3-4

Pattern matching facility

- cancellation 3-6
- case command 7-16
- characters 3-5
- description 3-5
- expr command argument effect 7-26
- grep command 4-15
- limitations 7-2
- metacharacter See Metacharacter

redirection restriction 7-4

shell function 7-2

variable assignment, not applicable
7-8

Pattern

- grammar 7-38
- metacharacter 7-40

Percentage sign (%), BC modulo operator

symbol 8-3

Period (.)

ed use See Ed

filename use 3-3

pattern matching facility restrictions
7-2

vi See Vi

working directory change 4-11

Permission types 4-12

Permission

block special device notation 4-11

change 3-2

denial notation 4-12

directory permission

assignment 3-2

change 3-2

change 4-13

combinations designated 4-12

file creation, deletion notation 4-
12

file listing notation 4-12

notation 4-11

search notation 4-12

search permission 4-14

write permission 3-2

execute notation 4-12

file permission

change 3-1

denial notation 4-12

Permission (*continued*)
 file permission (*continued*)
 execute permission 4-12
 file creation, deletion notation 4-12
 file listing notation 4-12
 file protection 3-1
 notation 4-11
 read notation 4-12
 required 3-1
 write notation 4-12
 listing 4-11
 notation 4-11
 read notation 4-12
 search notation 4-12
 symbols designated 4-11
 user class specification 4-13
 write notation 4-12
 PHONE shell procedure 7-36
 PID
 \$\$ variable 7-10
 process identification number 4-18
 process identification number 4-19
 Pipe
 compose escapes See Mail
 file interchange 7-33
 function 3-9
 lpr command 4-20
 procedure 3-9
 symbol (|) 3-9
 symbol (|) 7-40
 Pipeline
 command list 7-14
 C-shell use See C-shell
 defined 3-9
 defined 7-13
 description 7-5
 DISTINCT1 shell procedure 7-33
 filter 7-5
 grammar 7-38
 notation designated 7-5
 procedure 7-5
 Plus sign (+)
 BC
 addition operator symbol 8-3
 unary operator symbol 8-12
 mail, message advancement 6-25
 mail, message advancement 6-8
 variable conditional substitution 7-28
 Positional parameter
 description 7-7
 direct access 7-12
 null value assignment 7-27
 number yield, \$# variable 7-9
 parameter substitution 7-8
 positioning 7-7
 prefix (\$) 7-8
 setting 7-7
 variable assignment statement positioning 7-7

pr command 4-20
 ~print escape See Mail
 Print working directory See pwd command
 Printing
 command See lpr command
 command See p command
 command See pr command
 ed See Ed
 mail See Mail
 Process identification number See PID
 Process
 background See Background process
 defined 7-1
 foreground See Foreground process
 number See PID
 status
 status 4-19
 .profile file
 description, use 7-10
 PATH variable setting 7-9
 variable export 7-9
 Program stopping 2-3
 Prompt character 2-1
 Prompt character 4-1
 ps command 4-18
 ps command 4-19
 C-shell use See C-shell
 PS1 variable 7-9
 PS2 variable 7-9
 pwd command 4-10
 pwd command 4-8

Q

q command
 ed exit See Ed
 mail
 exit 4-21
 exit 6-13
 exit 6-26
 exit 6-3
 exit 6-7
 message abortion 6-21
 q! See Vi
 Question mark (?)
 directory name, use avoidance 7-3
 ed use See Ed
 filename, use avoidance 3-4
 mail
 command summary printing 6-25
 compose escape listing 6-16
 compose escape listing 6-2
 compose escape listing 6-9
 help command 6-10

Question mark (?) (*continued*)
 mail (*continued*)
 help command 6-3
 metacharacter 7-2
 metacharacter 7-40
 pattern matching See metacharacter

 pattern-matching functions 3-6
 single character representation 3-6
 variable conditional substitution 7-28

quiet option See Mail

quit command
 See also q command
 BC exit 8-1
 BC exit 8-2
 ~quit escape See Mail

QUIT key, background process immunity 7-14

Quit See q command

Quotation marks, back ("")
 command line substitution 7-6
 command substitution 7-3
 command substitution 7-6
 quoting 7-40

Quotation marks, double (
 Quotation marks, double (
 Quotation marks, double (
 Quotation marks, double (
 Quotation marks, double (
 Quotation marks, double (
 Quotation marks, single (")
 C-shell use See C-shell
 filename, use avoidance 3-4
 grep command 4-15
 metacharacter escape 7-3
 pattern matching cancellation 3-6
 trap command 7-20
 variable substitution inhibition 7-8

Quoting
 backslash (\) use 7-40
 metacharacter escape 7-3
 quotation marks, back ("") use 7-40
 quotation marks, double (7-40

R

r character, read permission notation 4-12

R command See Mail

r command
 ed use See Ed
 mail use See Mail

-r option 3-7

-R option, recursive listing 4-9
 mail 6-23

read command
 exit status 7-22
 shell built-in command 7-29

read command (*continued*)
 special shell command 7-22
 vi See Vi

~read escape See Mail

Read See r command

Read-ahead 2-3

readonly command
 description 7-22
 shell built-in command 7-29
 special shell command 7-22

Record string See Mail

Redirection
 argument location 7-6
 case command 7-19
 cd arg command 7-22
 control command 7-19
 diagnostic output 7-4
 file descriptor 7-27
 for command 7-19
 if command 7-19
 input redirection
 procedure 3-8
 symbol (<) 3-8
 minus sign (-) effect 7-26
 output redirection 4-5
 output redirection
 symbol (>) 3-8
 pattern matching use restriction 7-4
 simple command line, appearance 7-13
 special character use restriction 7-4
 special shell command, restriction 7-21
) 2-2
) 7-40
 until command 7-19
 while command 7-19

Reference Manual
 directory removal information 4-10
 linking information 4-8
 sort command information 4-15
 stty information 4-3
 terminal characteristics setting 2-3

Regular expressions See Ed

rehash command
 C-shell use See C-shell

Relative pathname See Pathname

Reminder service
 automatic 4-22
 mail 6-23

Remove directory See rmdir command

Remove See rm command

Removing a directory 4-9

Renaming a file 4-6

Repeat command
 see Vi 5-32

reply command See Mail

Report option See Vi

Reserved word listing 7-40

Return code See \$? variable

RETURN key
 BC 8-1
 command execution 2-1
 command execution 4-3
 command-line buffer submittal 3-7
 mail, message display 4-21
 rm command 2-2
 rm command 4-6
 rmdir command 4-9
 RUBOUT key, program stopping 2-3

S

s command
 ed use See Ed
 mail 6-12
 mail 6-13
 mail 6-27
 -s option
 mail, subject specification 4-20
 mail, subject specification 6-23
 shell invocation 7-28
 scale command 8-5
 Scale See BC
 Screen See Scrolling screen
 Screen See Terminal screen
 Screen-oriented commands See Vi 5-9
 Scripts See Ed
 Scripts See Shell
 more 4-4
 Scrolling, control 4-3
 stopping 4-3
 se command See set command
 Search permission See Permission
 Search See Ed
 Search strings
 example designated 3-7
 Searching for a file 4-7
 Searching See / command
 Searching See Vi
 vi procedure See Vi
 sed command See Ed
 Semaphore 4-12
 Semicolon (;)
 BC, statement separation 8-13
 BC, statement separation 8-2
 case command break 7-16
 case delimiter symbol 7-40
 command list 7-13
 command separation 3-7
 command separator symbol 7-40
 C-shell use See C-shell
 ed use See Ed
 set all See Vi
 set command
 C-shell

set command (*continued*)
 C-shell (*continued*)
 variable value assignment 10-2
 mail
 description, use 6-15
 description, use 6-27
 option control 6-29
 name-value pair listing 7-12
 positional parameters setting 7-7
 shell built-in command 7-29
 shell flag setting 7-11
 special shell command 7-21
 sh command
 description 7-1
 mail 6-15
 mail 6-25
 mail 6-27
 shell invocation 7-12
 Shell command
 executing while in vi 5-11
 SHELL string 6-22
 SHELL string 6-30
 Shell
 argument passing 7-12
 command interpretation 3-6
 command
 search procedure 7-2
 special command See special command
 compose escapes See Mail
 conditional capability 7-14
 procedure 7-1
 description 7-1
 -e option 7-23
 echo command 4-15
 entry, mail mode source 6-16
 escape
 ed procedure See Ed
 mail procedure See Mail
 execution
 flag See option
 sequence 7-13
 termination 7-18
 exit
 -e option 7-23
 mail mode return 6-16
 procedure 7-18
 -t option 7-24
 function 7-1
 grammar 7-38
 in-line input document handling 7-26
 interactive 7-28
 interruption procedure 7-19
 invocation
 option 7-28
 procedure 7-12
 -k option 7-24
 mail
 invocation 6-4

Shell (*continued*)
 mail (*continued*)
 shell commands 6-15
 -n option 7-24
 option
 See also Specific Option
 designated, use 7-23
 setting 7-11
 pattern matching facility See Pattern
 matching facility
 positional parameter See Positional
 parameter
 procedure
 See also Specific Shell Procedure

 advantages over C programs 7-
 23
 byte access reduction consideration
 7-30
 creation 7-22
 description 7-2
 directory 7-23
 efficiency analysis 7-29
 efficiency awareness 7-29
 examples designated 7-31
 filter order consideration 7-
 30
 option See option
 scripts designation 7-31
 time command 7-29
 writing strategies 7-29
 redirection ability 7-4
 scripts See procedure
 special command
 See also Specific Special Command

 designated 7-21
 redirection restriction 7-21
 special shell variable 7-13
 state 7-10
 string See SHELL string
 -t option 7-24
 TERM variable See TERM variable

 -u option 7-24
 -v option 7-11
 variable See Variable
 -x option 7-11
 shift command
 argument processing 7-12
 shell built-in command 7-29
 si command See Mail
 Simple command See Command
 Single quotation marks See Quotation
 marks, single (")
 Slash (/)
 absolute pathname significance 3-4
 BC, division operator symbol 8-3
 command prepending suppression
 7-2

Slash (/) (*continued*)
 ed use See Ed
 pathname significance 3-4
 search command See Vi
 so command See Mail
 sort command 4-15
 Special character See Metacharacter
 ed use See Ed
 pattern matching facility 7-2
 Special characters
 designated 3-5
 pattern matching 3-5
 Special file
 description 3-1
 Sshared data file 4-12
 Standard error file See Output
 Standard error output See Error output
 Standard input file See Input
 Standard output file See Output
 Star (*)
 See also Asterisk (*)
 ed metacharacter See Ed
 Status
 command See ps command
 information procedures 4-18
 String option See Mail
 String variable 7-7
 searching for See Search
 stty command 4-3
 terminal setting 2-3
 Subdirectory 4-11
 ~subject escape See Mail
 Subshell, directory change 7-10
 Substitution command See s command
 Subtraction See BC
 Subtraction See Calculation
 Switch See Option
 defined 3-7
 regulations See Option
 System
 basic concepts 3-1
 characteristics 1-1
 composition 1-1
 mailbox See Mailbox
 tree-structured directory system 3-2

T

t command
 ed use See Ed
 mail 6-12
 mail 6-27
 mail 6-9
 -t option, shell procedure 7-24
 Table command See Ed
 Tabs
 ed See Ed
 tail command 4-5

Index

tbl command See Ed
 Temporary file
 directory (/tmp) 4-18
 kill command warning 4-18
 trap command, removal 7-20
 use recommendation 7-10
 term option See Vi
 TERM variable, changing 4-2
 Terminal screen
 output See Output
 scrolling screen See Scrolling screen

 Terminal
 changing 4-2
 name designation 2-2
 options setting 4-3
 resetting 2-3
 strange behavior remedy 2-3
 writing to See write command
 Terminals
 supported 4-3
 terse option See Vi
 test command
 argument 7-25
 brackets ([]) use in lieu of 7-24
 description, use 7-24
 operators 7-25
 options 7-24
 shell built-in command 7-29
 Text editor
 ed See Ed
 ex See Ex
 vi See Vi
 TEXTFILE shell procedure 7-37
 then clause See if command
 Tilde escape See Mail
 time command 7-29
 /tmp directory 4-18
 contents 3-4
 ~to escape See Mail
 Top command See t command
 Toplines option See Mail
 Toplines string See Mail
 Transfer command See t command
 trap command
 description, use 7-19
 implementation method 7-21
 multiple traps 7-21
 special shell command 7-21
 temporary file removal 7-20
 troff See Ed
 true command 7-26
 tty, terminal system name 2-2
 Type-ahead 2-3
 Type-ahead 4-3

Typing error correction 2-3

Index

U

u command See vi 5-30
 ed use See Ed
 mail 6-13
 mail 6-27
 mail 6-6
 vi See Vi
 -u option, shell procedure 7-24
 -u option
 mail 6-23
 ugo, permission classification 4-13
 umask command
 description 7-22
 directory permission change 3-2
 shell built-in command 7-29
 special shell command 7-22
 Undo command See u command
 Undo command See Vi
 undo command See Vi 5-30
 unset command See Mail
 until command
 continue command effect 7-17
 description, use 7-16
 exit status 7-16
 redirection 7-19
 shell built-in command 7-29
 User classes 4-13
 User
 addition 2-1
 classification 4-13
 mail See Mail
 mailbox See Mailbox
 new user 2-1
 permission See Permission
 /usr directory
 contents 3-4
 /usr/bin directory
 /bin duplicate determination 7-32
 command search 7-2
 contents 3-4

V

v command
 ed use See Ed
 mail 6-15
 mail 6-27
 mail 6-5
 -v option, input line printing 7-11
 -v option
 function 3-7
 Value See \$? variable
 Variable
 \$- variable 7-10
 \$? variable 7-9

Variable (*continued*)

- assignment
 - line command 7-6
 - string value 7-7
 - BC variable See BC
 - command environment composition 7-11
 - conditional substitution 7-27
 - description 7-7
 - double quotation marks (7-8
 - enclosure 7-8
 - execution sequence 7-7
 - expansion 7-3
 - export 7-9
 - expr command 7-26
 - file creation 7-19
 - global check 7-24
 - HOME See HOME variable
 - IFS See IFS variable
 - keyword parameter 7-11
 - listing procedure 7-11
 - MAIL See MAIL variable
 - name defined 7-7
 - null value assignment procedure 7-27
 - PATH See PATH variable
 - positional parameter See Positional parameter
 - prefix (\$) 7-8
 - PS1 See PS1 variable
 - PS2 See PS2 variable
 - set variable defined 7-27
 - special variable 7-9
 - string value assignment 7-7
 - substitution
 - double quotation marks (7-8
 - notation 7-40
 - redirection argument 7-4
 - single quotation marks (") 7-8
 - space interpretation 7-8
 - u option effect 7-24
 - test command 7-24
 - types designated 7-8
- Vertical bar (|)
- mail escape 6-19
 - or-if operator symbol (|) 7-13
 - pipe symbol 3-9
 - pipeline notation 7-5

Vi

- . command
 - See dot (.) command
- / command
 - searching 5-7
- O command
 - cursor movement 5-4
- appending text
 - a 5-16
 - See also inserting text
- args command 5-34
- b command, cursor movement 5-4
- breaking lines 5-20

Vi (*continued*)

- buffers
 - delete 5-25
 - delete See delete buffer
 - naming 5-18
 - selecting 5-18
- C command 5-23
- C shell
 - prompt 5-37
 - TERM variable 5-38
 - terminal type setting 5-38
- canceling changes 5-32
- caret (^), pattern matching 5-29
- caret (^), pattern matching 5-30
- cc command 5-23
- CNTRL-B
 - scrolling 5-5
- CNTRL-D
 - scrolling 5-5
 - subshell exit 5-36
- CNTRL-F
 - scrolling 5-5
- CNTRL-G
 - file status information 5-35
 - file status information 5-8
- CNTRL-J, inserting 5-19
- CNTRL-L
 - screen redraw 5-36
- CNTRL-Q, inserting 5-19
- CNTRL-S, inserting 5-19
- CNTRL-U
 - deleting an insertion 5-21
 - scrolling 5-5
- CNTRL-V, use 5-19
- co (copy) command 5-18
- colon (:)
 - line-oriented command, use 5-9
 - status line prompt 5-9
- command mode
 - cursor movement 5-4
 - entering 5-3
- command
 - line-oriented See line-oriented commands 5-9
 - repeating, dot (.) use 5-5
 - screen-oriented See screen-oriented commands 5-9
- control characters, inserting 5-19
- copying lines 5-18
- correcting mistakes 5-16
- crash, recovery 5-36
- current line
 - deleting 5-20
 - deleting 5-5
 - designated 5-2
 - line containing cursor 5-3
 - number, finding out 5-18
- cursor movement
 - \$ key 5-14
 - b 5-13

Vi (continued)

cursor movement (continued)

- backward 5-14
- BKSP 5-13
- by character 5-13
- by lines 5-14
- by words 5-13
- CNTRL-N 5-14
- CNTRL-P 5-14
- down 5-13
- down 5-4
- e 5-13
- F 5-13
- forward 5-14
- h 5-13
- H 5-14
- j 5-13
- j 5-14
- k 5-13
- k 5-14
- keys 5-4
- l 5-13
- L 5-14
- left 5-13
- left 5-4
- line beginning 5-4
- line end 5-4
- LINEFEED key 5-14
- lower left screen 5-4
- M 5-14
- RETURN key 5-14
- right 5-13
- right 5-4
- screen 5-14
- scrolling See scrolling 5-5
- See also scrolling
- SPACEBAR 5-13
- T 5-13
- to end of file 5-4
- up 5-13
- up 5-4
- upper left screen 5-4
- w 5-13
- word backward 5-4
- word forward 5-4
- right 5-13
- cw command 5-22
- D command 5-5
- d0 command 5-5
- date, finding out 5-11
- dd command 5-5
- delete buffer
 - use 5-25
- deleting text
 - by character 5-20
 - by line 5-20
 - by word 5-20
 - D 5-20
 - dd command 5-20
 - deleting an insertion 5-21

Vi (continued)

deleting text (continued)

- dw command 5-20
- methods 5-5
- repeating deletion 5-32
- undoing 5-30
- undoing deletion 5-4
- X command 5-20
- demonstration 5-1
- description 5-1
- dollar sign (\$)
 - cursor movement 5-4
 - pattern matching 5-29
 - use in line address 5-21
- dot (.) command See . command 5-5
- dot See also dot (.) command
- dot, use in line address 5-21
- dw command 5-5
- editing several files
 - changing the order 5-34
- end-of-line
 - displaying 5-39
- entering
 - at a specified line 5-12
 - at a specified word 5-13
 - procedure 5-2
 - with filename 5-12
 - with several filenames 5-33
- error messages
 - shortening 5-39
 - turning off 5-35
- ESCAPE, insert mode exit 5-3
- ESCAPE, insert mode exit 5-36
- exclamation point (!)
 - shell escape 5-11
- exiting
 - :q! 5-12
 - saving changes 5-32
 - saving file 5-10
 - temporarily 5-11
 - temporarily 5-34
 - without saving changes 5-32
 - :x 5-12
 - :x command 5-32
 - ZZ command 5-32

cat. no.
26-6402

RADIO SHACK

A Division of Tandy Corporation
USA: Fort Worth, Texas 76102
Canada: Barrie, Ontario L4M4W5

Tandy Corporation

Australia

91 Kurrajong Avenue
Mount Druitt, N.S.W. 2770

Belguim

Rue des Pieds D'Alouette, 39
5140 Naninne (Namur)

France

BP 147-95022
Cergy Pontoise Cedex

U.K.

Bilston Road Wednesbury
West Midlands WS10 7JN